

卓越工程师培养计划

·单片机·

http://www.phei.com.cn

陈忠平　编著



ATmega16 单片机C语言程序设计经典实例

電子工業出版社

Publishing House of Electronics Industry

北京 · BEIJING

内 容 简 介

本书以“项目为载体，采用任务驱动方式”编写而成，以 ATmega16 单片机为蓝本，结合 CodeVisionAVR、Proteus 虚拟仿真软件，从实验、实践、实用的角度，通过丰富的实例详细介绍 ATmega16 单片机 C 语言程序设计和单片机控制系统的应用技术。

本书的内容主要包括单片机及最小应用系统、单片机系统开发软件的使用、AVR 单片机 C 语言编程基础、LED 广告灯的设计、按键控制与 LED 数码管显示、中断控制的应用设计、定时器/计数器控制的应用设计、单片机串行通信设计、LED 点阵显示器的设计、LCD 显示设计、内置模拟比较器和 A/D 转换器的应用设计、SPI 串行总线的应用、TWI 串行总线的应用、D/A 转换器的应用和综合应用设计 15 个项目 89 个任务。

本书内容全面，取材新颖，叙述清楚，理论联系实际，突出实用特色。本书适合单片机爱好者自学使用，也可作为高等院校 AVR 单片机课程的“教、学、做”一体化的教学用书，也是从事单片机应用研发人员的实用参考书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

ATmega16 单片机 C 语言程序设计经典实例 / 陈忠平编著. —北京：电子工业出版社，2013.6
（卓越工程师培养计划）
ISBN 978-7-121-20309-1

Ⅰ. ①A… Ⅱ. ①陈… Ⅲ. ①单片微型计算机-C 语言-程序设计 Ⅳ. ①TP368.1 ②TP312

中国版本图书馆 CIP 数据核字（2013）第 093612 号

责任编辑：张剑（zhang@phei.com.cn）
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编　100036
开　　本：787×1092　1/16　印张：33　字数：845 千字
印　　次：2013 年 6 月第 1 次印刷
印　　数：4 000 册　　定价：79.00 元

凡所购买电子工业出版社的图书，如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

单片机又称为单片微型控制器，其实质是将一个计算机系统集成到一个芯片上。它具有体积小、质量轻、价格便宜、控制功能强等特点。现在单片机已经在很多领域中得到了广泛应用，但随着技术的进步和应用需求的不断发展，人们对控制器的综合性能要求也越来越高，也希望产品的性价比越来越高，特别是在一些中、低端应用领域，更期望通过集成度比较高，而且价格便宜的单片机来解决问题，主要体现在外设丰富、功能强大、运用灵活、速度快、功耗低、产品开发方便、容易升级等方面。

1997 年，由 Atmel 公司挪威设计中心的 A 先生与 V 先生利用 Atmel 公司的 Flash 新技术，共同研发出 RISC 精简指令集的高速 8 位 AVR 系列单片机。相对于出现较早也较为成熟的 51 系列单片机，AVR 系列单片机在功能、速度、功耗等方面具有独特的优势。

本书以 AVR 系列典型的 ATmega16 单片机为蓝本，结合 Proteus 仿真软件，以项目为载体，采用任务驱动方式的教学方法，通过丰富的 C 语言程序实例，由浅入深地介绍了 ATmega16 单片机的基础知识及各种应用开发技术。在编写过程中，编者注重题材的取舍，使本书具有以下 4 个特点。

【项目为载体，任务带动教学】本书以“项目为载体，采用任务驱动方式”编写，强调“教、学、做”一体，坚持理论知识够用的原则，并将知识点分散到多个任务中，使读者能够边学边做，轻松迈入单片机学习之旅。

【软件结合，虚拟仿真】沿用传统单片机学习与开发经验，通过相关编译软件（如 CVAVR）编写程序并生成*.Hex 与*.cof 文件，然后在 Proteus 中绘制好硬件电路图（这一过程相当于硬件电路的焊接），调用*.cof（或*.Hex）文件进行虚拟仿真（这一过程相当于硬件调试）。对于单片机初学者来讲，这样可以节约学习成本，提高学习积极性；对于单片机系统开发人员来讲，这样可以缩短开发时间，提高设计效率，降低开发成本。

【C 语言编程，增强可读性】C 语言是一种编译型程序设计语言，它兼顾了多种高级语言的特点，并具备汇编语言的功能。用 C 语言来编写程序会大大缩短开发周期，且明显地增加程序的可读性，便于改进和扩充。采用 C 语言进行单片机程序设计是单片机开发与应用的必然趋势。在传统的单片机技术学习方法中，均先学习汇编语言，然后再学习用 C 语言编写单片机程序代码，经过这种学习历程的工程技术人员深深地感悟到汇编指令太枯燥，学习起来费时费力，用汇编语言编写或读懂一个程序不是一件容易的事情。在现代教学过程中，使用 C 语言进行编程，不必对单片机的硬件结构有很深入的了解，编写程序相对简单，且程序的可读性和可移植性均很强。

【兼顾原理，注重实用】基本原理、基本实例一直是学习和掌握单片机应用技术的基本要求。本书侧重于实际应用，因此很少讲解相关的理论知识，这样就避免了知识的重复性。为紧随技术的发展，在编写过程中还注重知识的新颖性和实用性，因此本书中在讲解 SPI 总线、TWI 总线时均讲解了相应的外扩芯片技术，使读者学习的知识能够紧随时代的发展。

参加本书编写工作的有湖南工程职业技术学院陈忠平、徐刚强、李锐敏、周少华、龙晓庆、龚亮，湖南涉外经济学院侯玉宝、高金定，衡阳技师学院胡彦伦，湖南航天诚远精密机械有限公司刘琼，湖南科技职业技术学院高见芳，湖南三一重工集团王汉其等。全书由湖南工程职业技术学院陈建忠教授主审。由于编著者知识水平和经验的局限性，书中难免有错漏之处，敬请广大读者批评指正。编著者 E-mail:czpmcu@126.com

编著者

目　录

项目一　单片机及最小应用系统……1
任务 1　认识单片机……1
任务 2　AVR 列单片机的识读……6
任务 3　单片机最小系统及系统控制……16
任务 4　自制下载线……23
项目二　单片机系统开发软件的应用……26
任务 1　CVAVR 编译软件的使用……26
任务 2　AVR Studio 集成开发软件的使用……34
任务 3　PonyProg2000 下载软件的使用……39
任务 4　Proteus 7.10 仿真软件的使用……41
项目三　AVR 单片机 C 语言编程基础……50
任务 1　C 语言程序组成的识读……50
任务 2　运算符和表达式……60
任务 3　程序结构及流程控制……67
任务 4　数组与指针……77
任务 5　函数与编译预处理……84
项目四　LED 广告灯的设计……92
任务 1　LED 控制原理认知……92
任务 2　闪灯广告灯的设计……98
任务 3　流水广告灯的设计……102
任务 4　拉幕式与闭幕式广告灯的设计……106
任务 5　复杂广告灯的设计……110
项目五　按键控制与 LED 数码管显示……115
任务 1　按键控制原理认知……115
任务 2　LED 数码管显示原理认知……117
任务 3　单个 LED 数码管显示设计……121
任务 4　多个 LED 数码管显示设计……125
任务 5　条形 LED 花样选择显示……131
任务 6　查询式按键设计……134
任务 7　LED 数码管显示矩阵键盘按键的设计……137
任务 8　矩阵键盘按键控制 LED+条形 LED 的设计……144
任务 9　简单拨号键盘的设计……151
项目六　中断控制的应用设计……158
任务 1　中断控制原理……158
任务 2　带中断控制的 LED 花样灯……165

任务 3　采用外中断控制的条形 LED 彩灯设计……168
任务 4　采用两个外中断的条形 LED 彩灯控制……172
任务 5　采用两个外中断实现 LED 键控移位……176
任务 6　采用外中断实现计数……179
任务 7　采用外中断进行计数和清零控制……184
项目七　定时器/计数器控制的应用设计……189
任务 1　定时器/计数器控制原理认知……189
任务 2　用 T/C0 实现简单计数……205
任务 3　用 T/C0 实现流水灯控制……209
任务 4　用 T/C0 的 CTC 模式输出方波……212
任务 5　用 T/C0 的快速 PWM 模式实现调光控制……215
任务 6　用 T/C0 的相位修正 PWM 模式实现调光控制……220
任务 7　用 T/C1 计数功能实现数控分频……223
任务 8　用 T/C1 实现预置值计时器的设计……227
任务 9　用 T/C1 的 CTC 模式实现 LED 滚动闪烁控制……233
任务 10　用 T/C1 的快速 PWM 模式输出占空比可调方波……237
任务 11　用 T/C1 的相位修改 PWM 模式输出占空比可调方波……242
任务 12　用 T/C1 的输入捕捉模式测量低频脉冲周期……246
任务 13　用 T/C2 实现简单门铃控制……250
任务 14　用 T/C2 的异步计数功能实现 59s 计时器……254
项目八　单片机串行通信设计……260
任务 1　单片机串行通信原理认知……260
任务 2　甲机通过串口控制乙机 LED 显示状态……271
任务 3　甲机通过串口控制乙机计时……275
任务 4　单片机双机通信……283
任务 5　单片机自发自收数据……291
项目九　LED 点阵显示器的设计……298
任务 1　LED 点阵显示器控制原理认知……298
任务 2　一个 5×7 LED 点阵字符显示……300
任务 3　一个 8×8 LED 点阵字符串显示……303
任务 4　两个 8×8 LED 点阵字符串显示……310
任务 5　两个 8×8 LED 点阵滚动显示……317
任务 6　一个 16×16 LED 点阵汉字显示……321
任务 7　一个 16×16 LED 点阵汉字移位显示……327
任务 8　两个 16×16 LED 点阵汉字显示……330
任务 9　两个 16×16 LED 点阵汉字分批显示……336
任务 10　两个 16×16 LED 点阵汉字移位显示……340
项目十　LCD 显示设计……345
任务 1　LCD 显示原理认知……345
任务 2　字符式 LCD 显示字符串……351

任务 3　字符式 LCD 显示汉字……354
任务 4　字符式 LCD 显示可调数字钟……360
任务 5　字符式 LCD 滚动显示……366
任务 6　字符式 LCD 移位显示……369
项目十一　内置模拟比较器和 A/D 转换器的应用设计……375
任务 1　内置模拟比较器和 A/D 转换器……375
任务 2　系统电源电压监测器的设计……385
任务 3　低压报警器的设计……388
任务 4　简易电压表的设计……391
任务 5　电压监控器的设计……396
项目十二　SPI 串行总线的应用……401
任务 1　SPI 总线基础知识认知……401
任务 2　74LS164 驱动数码管显示可调数字钟……405
任务 3　74HC595 驱动 1 位数码管显示……413
任务 4　74HC595 驱动数码管动态显示……416
任务 5　74HC595 驱动 8 位数码管静态显示……421
任务 6　MAX7219 驱动数码管动态显示……427
项目十三　TWI 串行总线的应用……435
任务 1　TWI 总线基础知识……435
任务 2　24C256 开启次数读/写控制……444
任务 3　PCF857 驱动 8 位数码管显示……451
项目十四　D/A 转换器的应用……456
任务 1　DAC0832 D/A 转换……456
任务 2　TLC5615 D/A 转换……461
任务 3　MAX515 D/A 转换……467
项目十五　综合应用设计……471
任务 1　篮球计分器的设计……471
任务 2　按键选播电子音乐……481
任务 3　DS1302 时钟日历的设计……489
附录 A　Proteus 常用快捷键……500
附录 B　CodeVisionAVR 编译器常用库函数简介……501
参考文献……518

项目一　单片机及最小应用系统

〖知识目标〗
☺ 了解单片机的定义、类型、应用领域及当前主流单片机;
☺ 了解 AVR 单片机命名规则、封装形式;
☺ 掌握 ATmega16 单片机各个引脚的功能;
☺ 掌握 ATmega16 单片机最小系统的组成及相关电路的工作原理;
☺ 掌握 ATmega16 单片机基本接口电路的组成及电路功能。
〖能力目标〗
☺ 认识 ATmega16 单片机、晶体振荡器、RS-232 串口及电阻、电容等元器件;
☺ 学会搭建单片机最小系统应用电路;
☺ 了解单片机应用系统的开发过程。



任务 1　认识单片机

微型化是计算机的主要发展方向之一。在微型计算机中，单片微型计算机（简称单片机）是其重要的组成部分。单片机依靠一定的硬件基础，根据特定环境，完成特定的需求。目前，单片机已经渗透到日常生活的各个领域，几乎很难找到哪个领域没有单片机的踪迹。例如，导弹的导航装置，飞机上各种仪器和仪表的控制，计算机的网络通信与数据传输，工业自动化过程的实时监控与数据处理，广泛使用的各种智能 IC 卡，民用豪华轿车的安全保障系统，摄像机、全自动洗衣机的控制，以及无线遥控玩具、电子宠物等，这些都离不开单片机。更不用说自动控制领域的机器人、智能仪表、医疗器械了。所以，认识单片机，学习单片机，掌握单片机的开发与应用，是电子信息、计算机应用、通信工程、自动控制等专业领域工程技术人员必备的技能。

1．单片机的定义

单片机是单片微型计算机（Single Chip Microcomputer，SCM）的简称，它是在一块芯片上集成了中央处理部件（Central Processing Unit，CPU）、数据存储器（Random Access Memory，RAM）、程序存储器（Read Only Memory，ROM）定时器/计数器和多种输入/输出（I/O）接口等功能部件，片内各功能部件通过内部总线相互连接起来的微型计算机。

单片机特别适合于控制领域，因此国外大多数厂商、学者普遍将其称为微处理器（Micro Controller），缩写为 MCU（Micro Controller Unit），国内习惯性将其称为单片机。

2．单片机的特点

☺ 性价比高;

☺ 控制功能强;
☺ 高集成度、高可靠性、体积小;
☺ 低电压、低功耗;
☺ 易扩展。

3．单片机的应用领域

单片机已经渗透到人们生活的各个领域，在工业控制、智能仪表、家用电器、军事装置等方面都得到了极为广泛的应用。

【在智能仪器仪表中的应用】用单片机制作的仪器仪表，广泛应用于实验室、交通运输工具、计量等领域。能使仪器仪表数字化、智能化、多功能化，提高测试的自动化程度和精度，简化硬件结构，减少质量，缩小体积，便于携带和使用，同时降低成本，提高性能价格比。典型应用如数字式存储示波器、数字式 RLC 测量仪、智能转速表等。

【在工业控制方面的应用】在工业控制中，工作环境恶劣，各种干扰比较强，还需实时控制，这对控制设备的要求比较高。单片机由于集成度高、体积小、可靠性高、控制功能强，能对设备进行实时控制，所以被广泛应用于工业过程控制中，如电镀生产线、工业机器人、电动机控制、炼钢、化工等领域。

【在军事装置中的应用】利用单片机的可靠性高，适用温度范围宽，能工作在各种恶劣环境等特点，将其应用在航天航空导航系统、电子干扰系统、宇宙飞船等尖端武器、导弹控制、智能武器装置、鱼雷制导控制等方面。

【在民用电子产品中的应用】在民用电子产品中，目前单片机广泛应用于通信与各种家用电器，如手机、数码相机、MP3 播放机、智能空调等。

4．单片机的发展

自从 1974 年美国仙童（Fairchild）公司运用计算机技术生产了世界上第一块单片机（F8）以来，在短短的数十年中，单片机的发展主要经历了以下 4 个阶段。

☺ 初级阶段（1974—1976 年）：此阶段的单片机结构比较简单，控制功能比较单一。例如，仙童公司的 F8 系列单片机，只包含了中央处理器 CPU、64Kbit 的 RAM 和两个并行口，还需外接具有 ROM、定时器/计数器、并行接口的芯片。

☺ 低性能阶段（1976—1978 年）：以 Intel 公司的 MCS-48 系列为代表，其特点是采用专门的结构设计，但内部资源不够丰富。该系列的单片机片内集成了 8 位 CPU、并行 I/O 端口、8 位定时器/计数器、RAM、ROM 等。无串行 I/O 口，中断处理系统也比较简单，片内 RAM、ROM 容量较小，且寻址范围小于 4KB（B 为 Byte，即字节）。

☺ 高性能阶段（1978—1983 年）：以 Intel 公司的 MCS-51 系列为代表，它完善了外部总线，丰富了内部资源，并确立了单片机的控制功能。采用 16 位的外部并行地址总线，能对外部 64KB 的程序存储器和数据存储器空间进行寻址；还有 8 位数据总线及相应的控制总线，形成完整的并行三总线结构。同时还提供了多机通信功能的串行 I/O 端口，具有多级中断处理，16 位的定时器/计数器，片内的 RAM 和 ROM 容量增大，寻址范围可达 64KB。有的单片机片内还带有 A/D 转换接口、DMA 接口、PSW 等功能模块。

在 MCS-51 单片机指令系统中，增加了大量的功能指令，如在基本控制功能方面设置了大量的位操作指令，使它和片内的位地址空间构成了单片机所独有的布尔逻辑操作系统，增

强了单片机的位操作控制功能；还有许多条件跳转、无条件跳转指令，从而增强了指令系统的控制功能。在单片机的片内设置了特殊功能寄存器（Special Function Register，SFR），为外围功能电路的集中管理提供了方便。

☺ 8 位超高性能单片机的巩固发展及 16 位、32 位、64 位单片机的推出与发展阶段（1983 年至今）：超 8 位单片机增加了 DMA（Direct Memory Access）直接数据存储存取通道、特殊串行接口等。16 位单片机的 CPU 为 16 位，片内 RAM 和 ROM 的容量进一步增大，片内 RAM 为 232B，ROM 为 8KB，片内带有高速 I/O 部件，多通道 10 位 A/D 转换器，8 级中断处理功能，实时处理能力更强，并开发了片内带 FLASH 程序存储器（Flash Memory）等功能。

5．单片机的系列

按指令集的不同，单片机主要有复杂指令集（Complex Instruction Set Computing，CISC）结构和精简指令集（Reduced Instruction Set Computing，RISC）结构两大类。

采用 CISC 结构的单片机，其指令丰富，功能较强，但取指令和取数据不能同时进行，速度受限，价格偏高。

采用 RISC 结构的单片机，取指令和取数据能够同时进行，便于采用流水线操作，且大部分指令为单指令周期，运行速度快；同时，程序存储器的空间利用率高，有利于实现超小型化。

CISC 结构的单片机有 Intel MCS-51 系列、Motorola M68HC 系列、Atmel AT89 系列、STC89 系列、华邦公司的 W77/W78 系列等；RISC 结构的单片机有 Microchip PIC 系列、韩国三星 KS57C 系列 4 位单片机、中国台湾义隆 EM-78 系列、Atmel AVR 系列、NXP（原 Philips）P89LPC90 系列等。一般在控制关系较简单的小家电中可以采用 RISC 结构单片机；在控制关系复杂的场合应采用 CISC 结构单片机。

6．市场上的主流单片机

【80C51 单片机】应用最广泛的 8 位单片机首推 Intel 的 MCS-51 系列。因其硬件结构合理，指令系统规范，加之生产历史“悠久”，有先入为主的优势。由于 Intel 公司在嵌入式应用方面将重点放在 x86、奔腾等与 PC 类兼容的高档芯片的开发上，所以 Intel 公司以专利互换和专利出售的方式将 80C51 内核授权给其他多家集成电路制造商，如 Philips、NEC、Atmel、AMD、华邦公司等。拿到该授权的厂商，在原有基础上进行性能上的扩充，使得芯片得到进一步的完善，形成了一个庞大的体系，直到现在仍在不断翻新，把单片机世界炒得沸沸扬扬。这样，80C51 单片机就变成了众多芯片制造厂商支持的大家族，统称为 80C51 单片机，有时也简称 51 系列单片机。客观事实证明，80C51 已成为 8 位单片机的主流，成了事实上的标准 MCU 芯片。目前国内许多学校仍以它作为典型机来进行单片机的教学。

【NXP 单片机】2007 年 Philips 半导体公司更名为 NXP，在我国又称为恩智浦。NXP 公司的单片机是基于 80C51 内核的单片机，嵌入了掉电检测、模拟及片内 RC 振荡器等功能。与标准 8051 单片机相比，它还增加了一个定时器/计数器并提供两种工作模式，有些型号的单片机还提供了 I^2C 或 SPI 串行接口。NXP 单片机具有集成度高、成本低、低功耗等特点，在应用设计中可以满足多方面的性能要求。

【Motorola 单片机】Motorola 是世界上最大的单片机厂商，该公司生产的单片机品种

全、选择余地大、新产品多。从 M6800 开始，开发了众多的品种，4 位、8 位、16 位和 32 位的单片机都能生产。在 8 位机方面，有 M68HC05 系列和升级产品 M68HC08、M68HC11、M68HC12，其中 M68HC05 有 30 多个系列、200 多个品种，产量已超过 20 亿片。16 位机 M68HC16 也有 10 多个品种，32 位机 M683XX 也有数十个品种。Motorola 单片机的特点之一是在同样单片机种类的速度下所用的时钟频率较 Intel 类单片机低得多，因而使得高频噪声低、抗干扰能力强，更适合于工控领域及恶劣的环境。

【Atmel 单片机】Atmel 公司生产的单片机主要有两大类，即 CISC 架构的 AT89 系列和 RISC 架构的 AVR 系列单片机。

AT89 系列单片机是 Atmel 公司基于 Intel 公司的 MCS-51 系列单片机研发出来的与 MCS-51 兼容但性能高于 MCS-51 的单片机。它一问世就以其优良的性能和实惠的价格赢得了国内研究人员的广泛欢迎。

AVR 系列单片机是增强型 RISC 架构、内置 Flash 存储器的单片机，具有高速处理能力，在一个时钟周期内可执行复杂的指令，每 MHz 可实现 1 MIPS（Million Instructions Per Second，每秒处理的百万级的机器语言指令数）的处理能力。AVR 单片机工作电压为 2.7～6.0V，可以实现耗电最优化。AVR 单片机广泛应用于计算机外部设备、工业实时控制、仪器仪表、通信设备、家用电器、宇航设备等各个领域。

【STC 单片机】STC89 系列单片机是 MCS-51 系列单片机的派生产品。它在指令系统、硬件结构和片内资源上与标准 8052 单片机完全兼容。STC89 系列单片机具有高速（最高时钟频率 90MHz）运行、低功耗、在系统（In-System Program，ISP）/在应用可编程（In-Application Program，IAP）、不占用户资源等特点。与 AT89 系列单片机相比，二者的主要区别是：①下载方式不同，AT89 系列（如 AT89S51/S52）主要是并行下载方式，而 STC89C51/52 采用串口下载方式；②内存大小有所不同，AT89S51/52 的片内 RAM 为 128/256B，而 STC89C51/52 片内 RAM 为 512B；③部分特殊功能寄存器不同；④STC 单片机相对执行速度更快，功能更加强大。

【华邦（Winbond）单片机】W77/W78 系列 8 位单片机是华邦公司推出 CMOS 型单片机，其引脚和指令集与 8051 兼容，但每个指令周期只需要 4 个时钟周期，速度提高了 3 倍，工作频率最高可达 40MHz。同时增加了 Watch Dog Timer、6 组外部中断源、2 组 UART、2 组 Data Pointer 及 Wait State Control Pin。W77/W78 系列单片机具有性价比高、功能强、型号多，可以满足各种不同应用场合等特点。

【Microchip 单片机】Microchip 单片机是市场份额增长较快的单片机，主要产品是 PIC 16C 系列和 PIC 17C 系列 8 位单片机。CPU 采用 RISC 结构，PIC 16C 系列仅有 33 条指令，采用 Harvard 双总线结构，具有运行速度快、低工作电压、低功耗、较大的 I/O 直接驱动能力、价格低、一次性编程、体积小等特点，适用于用量大、档次低、价格便宜的产品，在办公自动化设备、消费电子产品、电讯通信、智能仪器仪表、汽车电子、金融电子、工业控制等领域都有广泛的应用。

【NEC 单片机】NEC 单片机自成体系，以 8 位单片机 78K 系列产量最高，也有 16 位、32 位单片机。其单片机内置高精度的环形振荡器，在−40～85℃时精度可达 0.1%，并且还内置上电复位清零电路、低电压侦测电路和独立源看门狗电路。16 位以上单片机采用内部倍频技术，以降低外时钟频率。有的单片机采用内置操作系统。

【东芝单片机】东芝单片机从 4 位机到 64 位机，门类齐全。4 位机在家电领域有很大市

场；8 位机主要有 870 系列和 90 系列等，该类单片机允许使用慢模式，采用 32kHz 时钟时，功耗降至 10μA 数量级。CPU 内部多组寄存器的使用，使得中断响应与处理更加快捷。东芝的 32 位单片机采用 MIPS 3000A RISC 的 CPU 结构，面向 DVD、数码相机、图像处理等市场。

【富士通单片机】为满足对灵活性日益增长的需求，富士通不仅提供 8 位和 16 位的单片机，同时还提供高性能的 32 位 RISC 控制器。8 位机使用的是 16 位机的 CPU 内核，也就是说 8 位机与 16 位机所用的指令相同，使得开发比较容易。8 位单片机有著名的 MB8900 系列，16 位机有 MB90 系列。富士通单片机具有加密性强、快捷的运算能力、强大的抗电磁干扰能力、在线烧录和在线调试等特点，主要应用于汽车、工业、通信、消费电子和计算机外围设备。

【LG 公司的 GMS90 系列单片机】LG 公司生产的 GMS90 系列单片机与 Intel MCS-51 系列，Atmel 89C51/52、AT89C2051 等单片机兼容，采用 CMOS 技术，高达 40MHz 的时钟频率，应用于多功能电话、智能传感器、电表、工业控制、防盗报警装置、计费器、IC 卡装置、DVD、VCD、CD-ROM 等。

【凌阳 16 位单片机 SPCE】中国台湾凌阳公司推出的内核为 16 位微处理器的芯片，即 SPCE 系列单片机。该系列主要有集成度高、体积小、可靠性高、低功耗、低电压、性价比高、适合各种控制应用等特点。如 SPCE061A 型单片机，内嵌 32KB 的 Flash 存储器，具有 DPS 功能，有很强的信息处理能力，最高时钟可达 49MHz，尤其适用于数字语音播报与识别等应用领域，是数字语音识别与语音信息处理的理想产品。

【Scenix 单片机】Scenix 公司推出的 8 位 RISC 结构 SX 系列单片机在技术上有其独到之处。SX 系列单片机只有双时钟设置，指令运行速度可达 50/75/100MIPS；具有虚拟外设功能，柔性化 I/O 端口，所有 I/O 端口都可单独编程设定，提供各种 I/O 的库函数，用于实现各种 I/O 模块的功能，如多路 UART、多路 A/D、PWM、SPI、DTMF、FS、LCD 驱动等。采用 E^2PROM/Flash 程序存储器，可以实现在线系统编程。通过计算机 RS-232C 接口，采用专用串行电缆，即可对目标系统进行在线实时仿真。

【EPSON 单片机】EPSON 单片机以低电压、低功耗和内置 LCD 驱动器等特点著称，尤其是 LCD 驱动部分做得很好，广泛用于工业控制、医疗设备、家用电器、仪器仪表、通信设备和手持式消费类产品等领域。目前，EPSON 已推出 4 位单片机 SMC62 系列、SMC63 系列、SMC60 系列和 8 位单片机 SMC88 系列。

【三星单片机】三星单片机有 KS51 和 KS57 系列 4 位单片机、KS86 和 KS88 系列 8 位单片机、KS17 系列 16 位单片机和 KS32 系列 32 位单片机。三星单片机为 OTP 型（One Time Programmable，一次性可编程）ISP 在片编程功能。三星公司在单片机技术上以引进消化发达国家的技术、生产与之兼容的产品，然后以价格优势取胜。例如，在 4 位机上采用 NEC 的技术，8 位机上引进 Zilog 公司 Z8 的技术，在 32 位机上购买了 ARM7 内核，还有 DEC 的技术、东芝的技术等。

【MSP430 单片机】MSP430 系列单片机是美国德州仪器公司（TI）1996 年开始推向市场的一种 16 位超低功耗、具有精简指令集（RISC）的混合信号处理器（Mixed Signal Processor）。该系列单片机多应用于需要电池供电的便携式仪器仪表中。

7. 学习单片机的方法

【熟悉和了解单片机的内部和外部资源】因为单片机芯片的内部及外部资源都需要开发者自己管理，且在开发过程中，根据实际需求，开发者还得自己设计单片机外围电路，所以

需要熟悉和了解一定的硬件知识。

【了解并掌握单片机指令系统】开发者是通过程序对单片机进行相应的控制，而这些程序是通过相应的指令来编写的。单片机的指令系统主要有汇编语言或单片机 C 语言，开发者可以只使用其中一种，或者使用这两种语言进行程序的编写。由于单片机 C 语言具有编程和调试灵活方便，生成的代码编译效率高，完全模块化，可移植性好，便于项目维护管理，可以直接操作单片机硬件等特点，所以现在许多单片机程序开发者均采用单片机 C 语言进行单片机系统的开发。

【了解并熟练掌握常用软件的使用方法】进行单片机系统开发时，需要用到许多软件，不要求开发者对每个软件都很熟悉，但对于常用软件的使用应该特别熟悉。例如，使用 Keil C51 进行源程序的编译与调试，使用 Proteus 软件进行硬件仿真，单片机的程序固化软件的使用等。

【坚持手、脑并用的原则】在单片机的学习过程中，一定要坚持手、脑并用的原则，多做、多看、多想，先看别人编写的程序，再学习修改别人的程序并仿写类似的程序，最后自己设计、编写程序，也就是“看程序→仿程序→写程序”的过程，做到“实践、实践、再实践”。

任务 2　AVR 列单片机的识读

基于 8051 内核的微控制器采用了复杂指令系统 CISC（Complex Instruction Set Computing）体系。CISC 结构的单片机是传统的冯•诺依曼（Von-Neumann）结构，这种结构又称为普林斯顿（Princeton）体系结构，其片内程序空间和数据空间合在一起，取指令和操作数都是通过同一簇总线分时进行的。当高速运算时，取指令和操作数不能同时进行，否则将会造成传输通道上的瓶颈现象。

1997 年，Atmel 公司挪威设计中心的 A 先生和 V 先生出于市场需求的考虑，推出了增强型内置 Flash 的 8 位精简指令集微处理器（Reduced Instruction Set CPU，RISC），并将其命名为 AVR。

RISC 结构的 AVR 单片机是新型的哈佛（Harvard）结构，采用双总线结构。AVR 单片机的快速存取寄存器文件由 32 个通用寄存器组成。传统的基于累加器的结构需要大量的程序代码，以实现累加器和存储器之间的数据传送。在 AVR 单片机中，32 个寄存器全部直接与运算逻辑单元（ALU）相连，每个通用工作寄存器都代替了累加器。这就使微处理器在前一条指令执行时，就取出现行的指令，从而可以避免传统 CISC 单片机的累加器和存储器之间数据传送时的瓶颈现象，提高了系统的性能。

1．AVR 单片机的主要特征

【内嵌高质量的 Flash 存储器，并具有高保密性】AVR 单片机内嵌高质量的 Flash 程序存储器，擦写方便，支持 ISP 和 IAP，便于产品的调试、开发、生产和更新。片内含有长寿命的 E^2PROM，可长期保存关键数据，避免断电丢失。片内大容量的 RAM 不仅能满足一般场合的使用，同时也能更有效地支持使用高级语言开发系统程序，并可像 MCS-51 单片机那样扩展外部 RAM。

Flash 程序存储器具有保密锁死（Lock）功能，并且 Flash 程序存储器深藏于芯片内

部，可以通过自编程（Self Programming）方式远程下载被加密的更新代码。

【具有真正的 I/O 端口，能正确反映 I/O 真实情况】AVR 单片机通用数字 I/O 端口的 I/O 特性与 PIC 单片机的 HI/LOW 输出及三态高阻抗 HI-Z 输入相似，同时可设定类似于 8051 单片机内部有上拉电阻的输入端功能。AVR 的 I/O 端口是真正的 I/O 端口，能正确反映 I/O 的真实情况。AVR 的 I/O 端口资源灵活、功能强大，可以单独设定为输入/输出，也可以设定（初始）高阻输入，具有驱动能力强（可省去功率驱动器件）等特性。

【高速度、低功耗，具有省电休眠（SLEEP）功能】AVR 单片机一条指令的执行速度可达 50ns（20MHz），而耗电则在 1μA～2.5mA 之间。由于它采用新型的哈佛（Harvard）结构，具有一级流水线的预取指令功能，即对程序的读取和数据的操作使用不同的数据总线，因此当执行某一指令时，下一指令被预先从程序存储器中取出，使得指令可以在每一个时钟周期内被执行。AVR 单片机具有多种省电休眠模式，且可宽电压运行（2.7～5.5V），抗干扰能力强，可降低一般 8 位机中的软件抗干扰设计工作量和硬件的使用量。AVR 单片机具有 6 种休眠功能，能够从低功耗模式迅速唤醒。

【具有多种独立的时钟分频器】AVR 单片机片内具备多种独立的时钟分频器，分别供 URAT、I^2C、SPI 使用。其中与 8/16 位定时器配合的具有多达 10 位的预分频器，可通过软件设定分频系数，提供多种档次的定时时间。AVR 单片机中的定时器/计数器（单）可双向计数形成三角波，再与输出比较匹配寄存器配合，生成占空比可变、频率可变、相位可变的方波的设计方法（即脉宽调制输出 PWM），令人耳目一新。

【定位于工业级产品】AVR 单片机具有大电流 10～20mA（输出电流）或 40mA（吸收电流），可直接驱动 LED、SSR 或继电器。具有看门狗定时器（WDT）安全保护功能，可防止程序跑飞，以提高产品的抗干扰能力。

【具有超功能精简指令】AVR 单片机具有 32 个通用工作寄存器，相当于 8051 单片机中的 32 个累加器，克服了单一累加器数据处理造成的瓶颈现象。片内含有 128B～8KB 个 SRAM，可灵活使用指令运算，适合用 C 语言进行程序开发，易学、易写、易移植，并且具有很高的代码效率。

【通信方便、快捷、可靠，支持多机通信】其增强性的高速同/异步串口，具有硬件产生校验码、硬件检测和校验帧错、两级接收缓冲、波特率自动调整定位（接收时）、屏蔽数据帧等功能，提高了通信的可靠性，方便程序编写，更便于组成分布式网络和实现多机通信系统的复杂应用。其串口功能大大超过 MCS-51/96 单片机的串口，加之 AVR 单片机高速，中断服务时间短，故可实现高波特率通信。

【面向字节的高速硬件串行接口 TWI、SPI】TWI 与 I^2C 接口兼容，具备 ACK 信号硬件发送与识别、地址识别、总线仲裁等功能，能实现主/从机的收/发全部 4 种组合的多机通信。SPI 支持主/从机等 4 种组合的多机通信。

【增强嵌入式系统的可靠性】AVR 单片机有自动上电复位电路、独立的看门狗电路、低电压检测电路 BOD，多个复位源（自动上/下电复位、外部复位、看门狗复位、BOD 复位），可设置的启动后延时运行程序，增强了嵌入式系统的可靠性。

AVR 单片机技术体现了单片机集多种器件（包括 Flash 程序存储器、看门狗、E^2PROM、同/异步串行口、TWI、SPI、A/D 模数转换器、定时器/计数器等）和多种功能（增强可靠性的复位系统、降低功耗抗干扰的休眠模式、品种多门类全的中断系统、具有输入捕获和比较匹配输出等多样化功能的定时器/计数器、具有替换功能的 I/O 端口……）于

一身，充分体现了单片机技术的从“片自为战”向“片上系统（SoC）”过渡的发展方向。

2．AVR 单片机的分类及主要性能

AVR 单片机系列齐全，可适用于各种不同场合的要求。它主要有低档的 Tiny 系列、中档的 AT90 系列和高档的 ATmega 系列的产品。

【低档的 Tiny 系列】低档的 Tiny 系列 AVR 单片机是专门为需要小型微控制的简单应用优化设计的，具有很高的性价比，主要有 8 个引脚的 Tiny11/12/13/15/45/85，14 个引脚的 Tiny24/44/84，20 个引脚的 Tiny26/261/461/861/2313 等。

【中档的 AT90 系列】中档的 AT90 系列 AVR 单片机主要有 20 个引脚的 AT90S1200/2313/2323/2343，28 个引脚的 AT90S4433，40 个引脚的 AT90S4414/4434/8515/8535 等。

【高档的 ATmega 系列】高档的 ATmega 系列 AVR 单片机主要有 28 个引脚的 ATmega8/48/88/168，40 个引脚的 ATmega16/32/162/644/8535，64 个引脚的 ATmega128/165/325/64/645/1281/2561，100 个引脚的 ATmega3250/6450/640/1280/2560 等。

AVR 单片机从 8 脚到 100 脚，还有各种不同封装供用户选择，表 1-1～表 1-3 列出了目前主流 AVR 单片机的各系列性能参数。

自 2002 年以来，Atmel 公司对 AVR 单片机产品线进行了调整，逐步停止了性能重叠的中档低功耗 AVR 单片机中 AT90 系列的单片机。目前，AT90 系列某些型号的单片机已经被性能更加优越的 Tiny 系列或 ATmega 系列相应的 AVR 单片机替代（见表 1-4），所以在实际开发中建议不要再使用该系列的单片机。

3．AVR 单片机的命名规则

下面以 ATmega48V-10A 为例，介绍 AVR 单片机型号的命名规则。ATmega48V-10A 型号标志说明如下。

☺ “V”代表低/宽电压版本，以新推出的 AVR（如 ATmega48/88/168，Tiny13/2313 等）产品来说，带“V”表示工作电压为 1.8～5.5V 范围；不带“V”表示工作电压为 2.7～5.5V 范围。对于一些较老型号的 AVR 的单片机而言，有用“L”来表示的，“L”是“Low”的缩写，带“L”表示它可以支持低电压，它支持的电压范围为 2.7～5.5V；不带“L”的则表示支持的电压范围为 4.5～5.5V。

☺ 后缀的数字部分代表单片机可以支持的最高系统时钟，如 ATmega48V-10A 指的是可以支持到 10MHz 的系统时钟，ATmega64-16AU 及 ATmega64-16AI 指的是可以支持到 16MHz 的系统时钟。

☺ 后缀的第 1 个字母代表单片机的封装形式，“P”表示 DIP 封装；“A”表示 TQFP 封装；“S”表示 SOIC 封装；“M”表示 MLF 封装。如 ATmega48V-10A，说明它是 TQFP 封装。

☺ 后缀的第 2 个字母代表单片机的应用级别，“C”表示商业级；“I”表示工业级；“A”表示汽车级；“E”表示扩展级（温度范围为-40～105℃）；“U”表示工业级，符合 ROHS。如 ATmega64L-8AC 带“C”为商业级，ATmega64L-8AI 带“I”为工业级。

> 🕮 说明：
> 由于现欧美要求使用无铅 IC，所以 ATMEL 未来将只推出带“U”的 AVR MCU，它将取代原来带“I”的型号，如 ATmge8L-8AU 将取代原来的 ATmega64L-8AI。

表 1-1　Tiny 系列 AVR 单片机的性能

型号	Flash/KB	E^2PROM/B	SRAM/B	最大I/O数	系统时钟/MHz	电压范围/V	8位定时器	16位定时器	PWM	SPI	US-ART	TWI	在线编程ISP	10位A/D通道	模拟比较器	看门狗	片内振荡器	中断数	外部中断数	自编程	普通封装形式	绿色封装形式
Tiny11	1	—	—	6	6	2.7～5.5	1	—	—	—	—	—	有	—	有	有	有	4	1	—	PDIP8,SOIC8	PDIP8,SOIC8
Tiny12	1	64	—	6	8	1.8～5.5	1	—	—	—	—	—	有	—	有	有	有	5	1	—	PDIP8,SOIC8	PDIP8,SOIC8
Tiny13	1	64	64	6	20	1.8～5.5	1	—	2	—	—	—	有	4	有	有	有	9	6	有	PDIP8,SOIC8,SSOIC8,MLF20	PDIP8,SOIC8,SSOIC8,MLF20
Tiny15L	1	64	—	6	1.6	2.7～5.5	2	—	1	—	—	—	有	4	有	有	有	8	1	—	PDIP8,SOIC8	
Tiny25	2	128	128	6	20	1.8～5.5	2	—	4	USI	—	USI	有	4	有	有	有	15	7	有		PDIP8,SOIC8,MLF20
Tiny45	4	256	256	6	20	1.8～5.5	2	—	4	USI	—	USI	有	4	有	有	有	15	7	有		PDIP8,SOIC8,MLF20
Tiny85	8	512	512	6	20	1.8～5.5	2	—	4	USI	—	USI	有	4	有	有	有	15	7	有		PDIP8,SOIC8,MLF20
Tiny28L	2	—	—	11	4	1.8～5.5	1	—	—	—	—	—	有	—	有	有	有	5	2	—	PDIP28,MLF32,TQFP32	PDIP28,MLF32,TQFP32
Tiny24	2	128	128	12	20	1.8～5.5	1	1	4	USI	—	USI	有	8	有	有	有	17	12	有		PDIP14,SOIC14,MLF20
Tiny44	4	256	256	12	20	1.8～5.5	1	1	4	USI	—	USI	有	8	有	有	有	17	12	有		PDIP14,SOIC14,MLF20
Tiny84	8	512	512	12	20	1.8～5.5	1	1	4	USI	—	USI	有	8	有	有	有	17	12	有		PDIP14,MLF20
Tiny26	2	128	128	16	16	2.7～5.5	2	—	2	USI	—	USI	有	11	有	有	有	11	1	—	PDIP20,MLF32,SOIC20	PDIP20,MLF32,SOIC20
Tiny261	2	128	128	16	20	1.8～5.5	1	1	5	USI	—	USI	有	11	有	有	有	19	18	有		PDIP20,MLF32,SOIC20
Tiny461	4	256	256	16	20	1.8～5.5	1	1	5	USI	—	USI	有	11	有	有	有	19	18	有		PDIP20,MLF32,SOIC20
Tiny861	8	512	512	16	20	1.8～5.5	1	1	5	USI	—	USI	有	11	有	有	有	19	18	有		PDIP20,MLF32,SOIC20
Tiny2313	2	128	128	18	20	1.8～5.5	1	1	4	USI	1	USI	有	—	有	有	有	8	2	有	PDIP20,MLF20,SOIC20	PDIP20,MLF20,SOIC20

表 1-2　ATmega 系列 AVR 单片机的性能

型号	Flash/KB	E^2PROM/B	SRAM/B	最大I/O数	系统时钟/MHz	电压范围/V	8位定时器	16位定时器	PWM	SPI	USA-RT	TWI	在线编程ISP	10位A/D通道	模拟比较器	看门狗	片内振荡器	中断数	外部中断数	自编程	普通封装形式	绿色封装形式
ATmega48	4	256	512	23	20	1.8～5.5	2	1	6	1+USART	1	有	有	8	有	有	有	26	26	有	PDIP28,MLF32,TQFP32	PDIP28,MLF32,TQFP32
ATmega8	8	512	1024	23	16	2.7～5.5	2	1	3	1	1	有	有	8	有	有	有	18	2	有	PDIP28,MLF32,TQFP32	PDIP28,MLF32,TQFP32
ATmega88	8	512	1024	23	20	1.8～5.5	2	1	6	1+USART	1	有	有	8	有	有	有	26	26	有	PDIP28,MLF32,TQFP32	PDIP28,MLF32,TQFP32
ATmega168	16	512	1024	23	20	1.8～5.5	2	1	6	1+USART	1	有	有	8	有	有	有	26	26	有	PDIP28,MLF32,TQFP32	PDIP28,MLF32,TQFP32
ATmega8535	8	512	512	32	16	2.7～5.5	2	1	4	1	1	有	有	8	有	有	有	20	3	有	PDIP40,MLF44,TQFP44,PLCC44	PDIP40,MLF44,TQFP44
ATmega16	16	512	1024	32	16	2.7～5.5	2	1	4	1	1	有	有	8	有	有	有	20	3	有	PDIP40,MLF44,TQFP44	PDIP40,MLF44,TQFP44
ATmega32	32	1K	2048	32	16	2.7～5.5	2	1	4	1	1	有	有	8	有	有	有	19	3	有	PDIP40,MLF44,TQFP44	PDIP40,MLF44,TQFP44
ATmega644	64	2K	4096	32	20	1.8～5.5	2	1	6	1+USART	1	有	有	8	有	有	有	31	32	有		PDIP40,MLF44,TQFP44
ATmega8515	8	512	512	35	16	2.7～5.5	1	1	3	1	1	—	有	—	—	有	有	16	3	有	PDIP40,MLF44,TQFP44,PLCC44	PDIP40,MLF44,TQFP44
ATmega162	16	512	1024	35	16	1.8～5.5	2	2	6	1	2	—	有	—	有	有	有	28	3	有	PDIP40,MLF44,TQFP44	PDIP40,MLF44,TQFP44
ATmega128	128	4K	4096	53	16	2.7～5.5	2	2	8	1	2	有	有	8	有	有	有	34	8	有	MLF64,TQFP64	MLF64,TQFP64
ATmega165	16	512	1024	54	16	1.8～5.5	2	1	4	1+USI	1	USI	有	8	有	有	有	23	17	有	MLF64,TQFP64	MLF64,TQFP64
ATmega325	32	1K	2048	54	16	1.8～5.5	2	1	4	1+USI	1	USI	有	8	有	有	有	23	17	有	MLF64,TQFP64	MLF64,TQFP64
ATmega64	64	2K	4096	54	16	2.7～5.5	2	2	8	1	2	有	有	8	有	有	有	34	8	有	MLF64,TQFP64	MLF64,TQFP64
ATmega645	64	2K	4096	54	16	1.8～5.5	2	1	4	1+USI	1	USI	有	8	有	有	有	23	17	有	MLF64,TQFP64	MLF64,TQFP64
ATmega1281	128	4K	8192	54	16	1.8～5.5	2	4	10	1+USART	2	有	有	8	有	有	有	48	17	有	—	MLF64,TQFP64

续表

型号	Flash/KB	E^2PROM/B	SRAM/B	最大I/O数	系统时钟/MHz	电压范围/V	8位定时器	16位定时器	PWM	SPI	USA-RT	TWI	在线编程ISP	10位A/D通道	模拟比较器	看门狗	片内振荡器	中断数	外部中断数	自编程	普通封装形式	绿色封装形式
ATmega2561	256	4K	8192	54	16	1.8～5.5	2	4	10	1+USART	2	有	有	8	有	有	有	48	17	有	—	MLF64,TQFP64
ATmega3250	32	1K	2048	69	16	1.8～5.5	2	1	4	1+USI	1	USI	有	8	有	有	有	32	17	有	TQFP100	TQFP100
ATmega6450	64	2K	4096	69	16	1.8～5.5	2	1	4	1+USI	1	USI	有	8	有	有	有	32	17	有	TQFP100	TQFP100
ATmega640	64	4K	8192	86	16	1.8～5.5	2	4	16	1+USART	4	有	有	16	有	有	有	57	32	有	—	TQFP100,BGA100
ATmega1280	128	4K	8192	86	16	1.8～5.5	2	4	16	1+USART	4	有	有	16	有	有	有	57	32	有	—	TQFP100,BGA100
ATmega2560	256	4K	8192	86	16	1.8～5.5	2	4	16	1+USART	4	有	有	16	有	有	有	57	32	有	—	TQFP100,BGA100

表 1-3　其他系列 AVR 单片机的性能

型号	Flash/KB	E^2PROM/B	SRAM/B	最大I/O数	系统时钟/MHz	电压范围/V	特殊特征	8位定时器	16位定时器	PWM	SPI	US-ART	TWI	在线编程ISP	10位A/D通道	模拟比较器	看门狗	片内振荡器	中断数	外部中断数	自编程	绿色封装形式
ATmega48P	4	256	512	23	16	1.8～5.5		2	1	6	1+USART	1	有	有	8	有	有	有	26	26	有	MLF32,TQFP32
ATmega88P	8	512	512	23	16	1.8～5.5		2	1	6	1+USART	1	有	有	8	有	有	有	26	26	有	MLF32,TQFP32
ATmega168P	16	1K	1024	23	16	1.8～5.5		2	1	6	1+USART	1	有	有	8	有	有	有	26	26	有	MLF32,TQFP32
ATmega164P	16	512	1024	32	20	1.8～5.5		2	1	6	1+USART	2	有	有	8	有	有	有	31	32	有	PDIP40,MLF44,TQFP44
ATmega324P	32	1K	2048	32	20	1.8～5.5		2	1	6	1+USART	2	有	有	8	有	有	有	31	32	有	PDIP40,MLF44,TQFP44
ATmega644P	64	2K	4096	32	20	1.8～5.5		2	1	6	1+USART	2	有	有	8	有	有	有	31	32	有	PDIP40,MLF44,TQFP44
ATmega165P	16	512	1024	54	16	1.8～5.5		2	1	4	1+USI	1	USI	有	8	有	有	有	23	17	有	MLF64,TQFP64
ATmega325P	32	1K	2048	54	16	1.8～5.5		2	1	4	1+USI	1	USI	有	8	有	有	有	23	17	有	MLF64,TQFP64
ATmega3250P	32	1K	2048	69	16	1.8～5.5		2	1	4	1+USI	1	USI	有	8	有	有	有	32	17	有	TQFP100
ATmega169P	16	512	1024	54	16	1.8～5.5	LCD 4×25	2	1	4	1+USI	1	USI	有	8	有	有	有	23	17	有	MLF64,TQFP64
ATmega329P	32	1K	2048	54	16	1.8～5.5	LCD 4×25	2	1	4	1+USI	1	USI	有	8	有	有	有	25	17	有	MLF64,TQFP64
ATmega3290P	32	1K	2048	69	16	1.8～5.5	LCD 4×40	2	1	4	1+USI	1	USI	有	8	有	有	有	25	32	有	TQFP100

表 1-4　替代的 AVR 单片机

原 来 型 号	可更换的新型号	原 来 型 号	可更换的新型号
AT90LS4433	ATmega81	AT90LS8535	ATmega8535
AT90S1200	Tiny2313	AT90S2313	Tiny2313
AT90S4433	ATmega8	AT90S8515	ATmega8515
AT90S8535	ATmega8535	ATmega103	ATmega128
ATmega161	ATmega162	ATmega103L	ATmega128L
ATmega161L	ATmega162L	ATmega323	ATmega32
ATmega163	ATmega16	ATmega323L	ATmega32L
ATmega163L	ATmega16L		

4．ATmega16 单片机封装形式

ATmega16 单片机的普通封装有 3 种形式，即 PDIP-40（Plastic Dual In-line Package，塑料双直列直插式）、TQFP-44（Thin Quad Flat Package，薄塑四角扁平封装）和 MLF-44（Micro Lead Frame，微引线框架封装），其外部封装形式如图 1-1 所示。

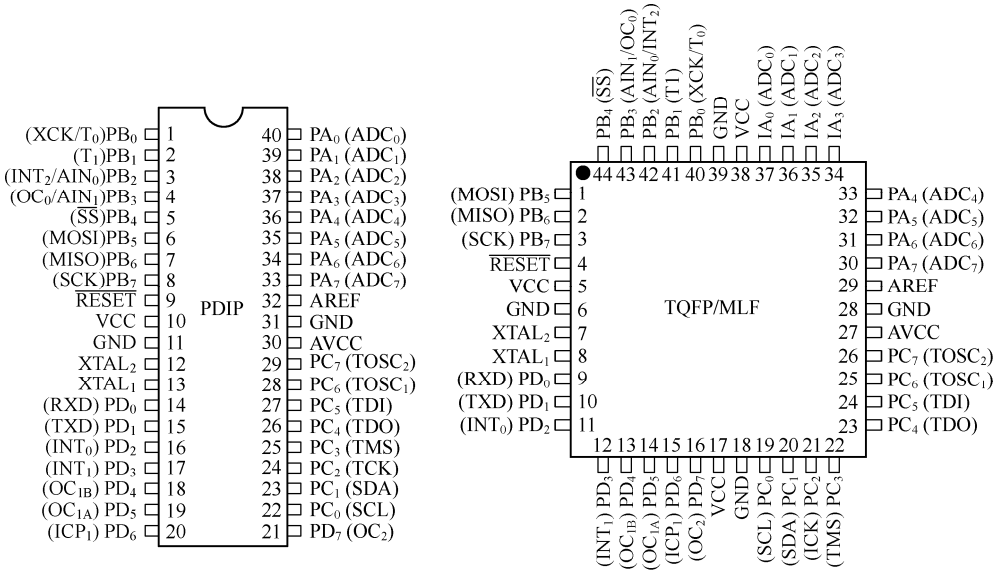


图 1-1　ATmega16 单片机的芯片封装

5．ATmega16 单片机引脚功能

PDIP-40（Plastic Dual In-line Package，塑料双直列直插式）封装的 ATmega16 单片机的外部引脚功能见表 1-5。

表 1-5　ATmega16 单片机的外部引脚功能

引　脚　号	引 脚 名 称	引 脚 功 能
1	(XCK/T_0) PB_0	XCK 为 USART 外部时钟输入；T_0 为定时器/计数器 0 的外部计数输入端；PB_0 为 PB 端口的第 0 位
2	(T_1) PB_1	T_1 为定时器/计数器 0 的外部计数输入端；PB_1 为 PB 端口的第 0 位

续表

引　脚　号	引 脚 名 称	引 脚 功 能
3	(INT_2/AIN_0) PB_2	INT_2 为外部中断 2 的输入端；AIN_0 为模拟比较器输入端 0；PB_2 为 PB 端口的第 2 位
4	(OC_0/AIN_1) PB_3	OC_0 为定时器/计数器 0 输出比较的匹配输出端；AIN_1 为模拟比较器输入端 1；PB_3 为 PB 端口的第 3 位
5	($\overline{SS}$)PB_4	$\overline{SS}$ 为 SPI 从机选择端；PB_4 为 PB 端口的第 4 位
6	(MOSI) PB_5	MOSI 为 SPI 主机输出、从机输入端；PB_5 为 PB 端口的第 5 位
7	(MISO) PB_6	MISO 为 SPI 主机输入、从机输出端；PB_6 为 PB 端口的第 6 位
8	(SCK) PB_7	SCK 为 SPI 串行时钟输入端；PB_7 为 PB 端口的第 7 位
9	$\overline{RESET}$	复位端（低电平有效）
10	VCC	数字电源端
11	GND	数字地
12	$XTAL_2$	振荡器输出端
13	$XTAL_1$	振荡器输入端
14	(RXD) PD_0	RXD 为 USART 数据接收端；PD_0 为 PD 端口的第 0 位
15	(TXD) PD_1	TXD 为 USART 数据发送端；PD_1 为 PD 端口的第 1 位
16	(INT_0) PD_2	INT_0 为外部中断输入 0；PD_2 为 PD 端口的第 2 位
17	(INT_1) PD_3	INT_1 为外部中断输入 1；PD_3 为 PD 端口的第 3 位
18	(OC_{1B}) PD_4	OC_{1B} 为定时器/计数器 1 输出比较 1B 的匹配输出端；PD_4 为 PD 端口的第 4 位
19	(OC_{1A}) PD_5	OC_{1A} 为定时器/计数器 1 输出比较 1A 的匹配输出端；PD_5 为 PD 端口的第 5 位
20	(ICP_1) PD_6	ICP_1 为定时器/计数器 1 的输入捕获端；PD_6 为 PD 端口的第 6 位
21	(OC_2) PD_7	OC_2 为定时器/计数器 2 输出比较的匹配输出端；PD_7 为 PD 端口的第 7 位
22	(SCL) PC_0	SCL 为 TWI 时钟线；PC_0 为 PC 端口的第 0 位
23	(SDA) PC_1	SDA 为 TWI 数据线；PC_1 为 PC 端口的第 1 位
24	(TCK) PC_2	TCK 为 JTAG 测试时钟；PC_2 为 PC 端口的第 2 位
25	(TMS) PC_3	TMS 为 JTAG 测试模式；PC_3 为 PC 端口的第 3 位
26	(TDO) PC_4	TDO 为 JTAG 测试模式；PC_4 为 PC 端口的第 4 位
27	(TDI) PC_5	TDI 为 JTAG 测试时钟；PC_5 为 PC 端口的第 5 位
28	($TOSC_1$) PC_6	$TOSC_1$ 为实时时钟引脚 1；PC_6 为 PC 端口的第 6 位
29	($TDSC_2$) PC_7	$TOSC_2$ 为实时时钟引脚 2；PC_7 为 PC 端口的第 7 位
30	AVCC	模拟电源端
31	GND	模拟地
32	AREF	A/D 转换参考电压
33	(ADC_7) PA_7	ADC_7 为 A/D 转换输入端 7；PA_7 为 PA 端口的第 7 位
34	(ADC_6) PA_6	ADC_6 为 A/D 转换输入端 6；PA_6 为 PA 端口的第 6 位
35	(ADC_5) PA_5	ADC_5 为 A/D 转换输入端 5；PA_5 为 PA 端口的第 5 位
36	(ADC_4) PA_4	ADC_4 为 A/D 转换输入端 4；PA_4 为 PA 端口的第 4 位
37	(ADC_3) PA_3	ADC_3 为 A/D 转换输入端 3；PA_3 为 PA 端口的第 3 位
38	(ADC_2) PA_2	ADC_2 为 A/D 转换输入端 2；PA_2 为 PA 端口的第 2 位
39	(ADC_1) PA_1	ADC_1 为 A/D 转换输入端 1；PA_1 为 PA 端口的第 1 位
40	(ADC_0) PA_0	ADC_0 为 A/D 转换输入端 0；PA_0 为 PA 端口的第 0 位

6．存储器结构

ATmega16 单片机片内集成了 Flash 程序存储器、SRAM 数据存储器和 E^2PROM 数据存储器。这 3 个存储器空间互相独立，物理结构也不相同。虽然它们的物理结构不同，但是这 3 个存储器的空间均采用线性的平面结构，如图 1-2 所示。

【Flash 程序存储器】ATmega16 具有 16KB 的在线编程 Flash，用于存放程序指令代码。因为所有的 AVR 指令为 16 位或 32 位，故而 Flash 组织成 8K×16 位的形式。Flash 程序存储器被分为两个区，即引导（Boot）程序区和应用程序区。

Flash 程序存储器至少可以擦写 10^4 次。ATmega16 的程序计数器（PC）为 13 位，因此可以寻址 8KB 的程序存储器空间。

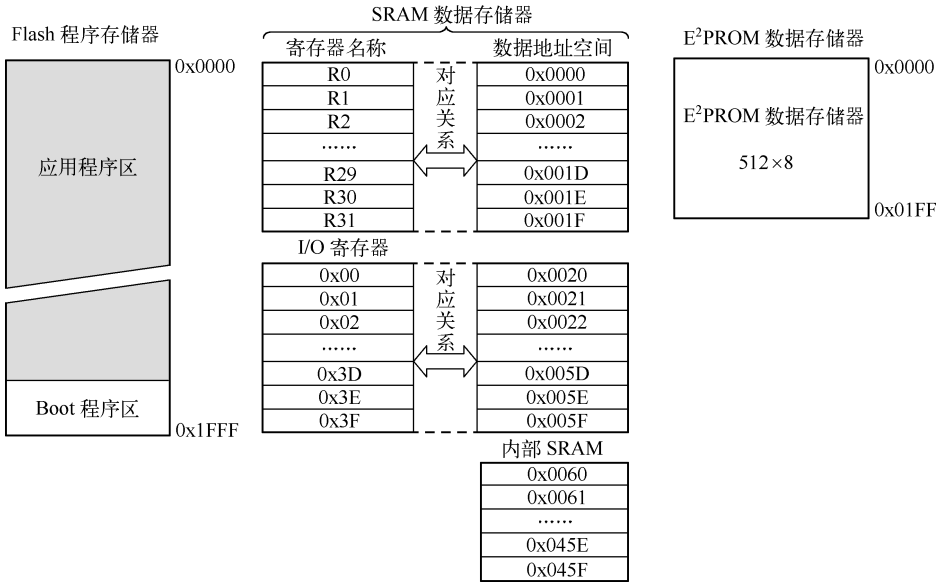


图 1-2　ATmega16 的存储器结构

【SRAM 数据存储器】ATmega16 具有 1120B 的片内 SRAM，由通用寄存器（R0～R31）、I/O 寄存器和数据存储器 SRAM 组成。

ATmega16 单片机 SRAM 的起始地址段 0x0000～0x001F（即 96 个地址）为寄存器文件（R0～R31 共 32 个 8 位通用寄存器），其中最后 6 个寄存器 R26～R31 组成 X、Y、Z 这 3 个 16 位寄存器，用于存放间接寻址的地址指针。R26 和 R27 组成 X 寄存器，R26 为 X 寄存器的低 8 位，R27 为 X 寄存器的高 8 位；R28 和 R29 组成 Y 寄存器，R28 为 Y 寄存器的低 8 位，R29 为 Y 寄存器的高 8 位；R30 和 R31 组成 Z 寄存器，R30 为 Z 寄存器的低 8 位，R31 为 Z 寄存器的高 8 位。

ATmega16 单片机 SRAM 的地址段 0x0020～0x005F 为 I/O 寄存器。I/O 寄存器支持专用的 I/O 指令访问和 SRAM 地址访问。如果使用 IN 和 OUT 指令访问 I/O 寄存器时，地址单元为 0x00～0x3F；而将 I/O 寄存器作为普通 SRAM 访问时，地址单元则为 0x0020～0x005F。

ATmega16 单片机的 I/O 寄存器的地址空间分配、名称的功能描述见表 1-6。

表 1-6　ATmega16 单片机的 I/O 寄存器的地址空间分配表

I/O（数据）地址	名　　称	功 能 描 述
0x00(0x0020)	TWBR	TWI 波特率寄存器
0x01(0x0021)	TWSR	TWI 状态寄存器
0x02(0x0022)	TWAR	TWI 从机地址寄存器
0x03(0x0023)	TWDR	TWI 数据寄存器
0x04(0x0024)	ADCL	ADC 数据寄存器低字节
0x05(0x0025)	ADCH	ADC 数据寄存器高字节
0x06(0x0026)	ADCSRA	ADC 控制和状态寄存器
0x07(0x0027)	ADMUX	ADC 多路选择器
0x08(0x0028)	ACSR	模拟比较控制和状态寄存器
0x09(0x0029)	UBRRL	USART 波特率寄存器低 8 位
0x0A(0x002A)	UCSRB	USART 波特率寄存器 B
0x0B(0x002B)	UCSRA	USART 波特率寄存器 A
0x0C(0x002C)	UDR	USART I/O 数据寄存器
0x0D(0x002D)	SPCR	SPI 控制寄存器
0x0E(0x002E)	SPSR	SPI 状态寄存器
0x0F(0x002F)	SPDR	SPI I/O 数据寄存器
0x10(0x0030)	PIND	端口 D 的外部输入引脚寄存器
0x11(0x0031)	DDRD	端口 D 的数据方向寄存器
0x12(0x0032)	PORTD	端口 D 的数据寄存器
0x13(0x0033)	PINC	端口 C 的外部输入引脚寄存器
0x14(0x0034)	DDRC	端口 C 的数据方向寄存器
0x15(0x0035)	PORTC	端口 C 的数据寄存器
0x16(0x0036)	PINB	端口 B 的外部输入引脚寄存器
0x17(0x0037)	DDRB	端口 B 的数据方向寄存器
0x18(0x0038)	PORTB	端口 B 的数据寄存器
0x19(0x0039)	PINA	端口 A 的外部输入引脚寄存器
0x1A(0x003A)	DDRA	端口 A 的数据方向寄存器
0x1B(0x003B)	PORTA	端口 A 的数据寄存器
0x1C(0x003C)	EECR	E^2PROM 控制寄存器
0x1D(0x003D)	EEDR	E^2PROM 数据寄存器
0x1E(0x003E)	EEARL	E^2PROM 地址寄存器低 8 位
0x1F(0x003F)	EEARH	E^2PROM 地址寄存器高 8 位
0x20(0x0040)	UBRRH	USART 波特率寄存器高 4 位
	UCSRC	USART 状态寄存器 C
0x21(0x0041)	WDTCR	看门狗定时控制寄存器
0x22(0x0042)	ASSR	异步模式状态寄存器
0x23(0x0043)	OCR2	定时器/计数器 2 输出比较寄存器
0x24(0x0044)	TCNT2	定时器/计数器 2 数据寄存器
0x25(0x0045)	TCCR2	定时器/计数器 2 控制寄存器
0x26(0x0046)	ICR1L	定时器/计数器 1 输入捕捉寄存器低 8 位

续表

I/O（数据）地址	名　称	功 能 描 述
0x27(0x0047)	ICR1H	定时器/计数器 1 输入捕捉寄存器高 8 位
0x28(0x0048)	OCR1BL	定时器/计数器 1 输出比较寄存器 B 低 8 位
0x29(0x0049)	OCR1BH	定时器/计数器 1 输出比较寄存器 B 高 8 位
0x2A(0x004A)	OCR1AL	定时器/计数器 1 输出比较寄存器 A 低 8 位
0x2B(0x004B)	OCR1AH	定时器/计数器 1 输出比较寄存器 A 高 8 位
0x2C(0x004C)	TCNT1L	定时器/计数器 1 数据寄存器低 8 位
0x2D(0x004D)	TCNT1H	定时器/计数器 1 数据寄存器高 8 位
0x2E(0x004E)	TCCR1B	定时器/计数器 1 控制寄存器 B
0x2F(0x004F)	TCCR1A	定时器/计数器 1 控制寄存器 A
0x30(0x0050)	SFIOR	特殊功能 I/O 寄存器
0x31(0x0051)	OSCCAL	内部 RC 振荡器校准值寄存器
	OCDR	在线调试寄存器
0x32(0x0052)	TCNT0	定时器/计数器 0 数据寄存器
0x33(0x0053)	TCCR0	定时器/计数器 0 控制寄存器
0x34(0x0054)	MCUCSR	MCU 控制和状态寄存器
0x35(0x0055)	MCUCR	MCU 控制寄存器
0x36(0x0056)	TWCR	TWI 控制寄存器
0x37(0x0057)	SPMCR	程序存储器写控制寄存器
0x38(0x0058)	TIFR	定时器/计数器中断标志寄存器
0x39(0x0059)	TIMSK	定时器/计数器中断屏蔽寄存器
0x3A(0x005A)	GIFR	通用中断标志寄存器
0x3B(0x005B)	GICR	通用中断控制寄存器
0x3C(0x005C)	OCR0	定时器/计数器 0 输出比较寄存器
0x3D(0x005D)	SPL	堆栈指针寄存器低 8 位
0x3E(0x005E)	SPH	堆栈指针寄存器高 8 位
0x3F(0x005F)	SREG	状态寄存器

直接给用户使用的 SRAM 单元为 0x006F～0x045F 共 1024B，这部分 SRAM 单元用于存放在程序执行过程中定义的各种变量。

【E^2PROM 数据存储器】ATmega16 包含 512B 的 E^2PROM 数据存储器。它是作为一个独立的数据空间而存在的，可以通过相关的寄存器按字节对其进行读/写操作。E^2PROM 的擦除次数达 10^5 次以上。E^2PROM 的访问由地址寄存器、数据寄存器和控制寄存器决定。



任务 3　单片机最小系统及系统控制

1．单片机最小系统

单片机最小系统又称为单片机基本系统，是指用最少的元器件能使单片机工作起来的一个最基本的应用系统。在这种系统中，使用 ATmega16 单片机的一些内部资源就能够满足

硬件设计需求，不需扩展外部的存储器或 I/O 接口等器件，通过用户编写的程序，单片机就能够达到控制的要求。

单片机的最小系统结构只能使用在控制较简单的场合，该系统包括单片机、时钟电路、复位电路等。同时，单片机要正常运行，还必须具备电源正常、时钟正常、复位正常 3 个基本条件。ATmega16 单片机组成的最小系统电路原理图如图 1-3 所示。

从图 1-3 中可以看出，电路以 ATmega16 单片机为核心，ATmega16 的第 12 脚和第 13 脚外接由 C1、C2 和 X1 构成的石英晶体振荡器电路（即时钟电路）；ATmega16 的第 9 脚外接由 R1、C3 构成的复位电路。当然，ATmega16 单片机要正常工作，还需提供电源，因此在实际电路中，ATmega16 的第 11 脚应该接地，而第 10 脚应该接电源+5V。

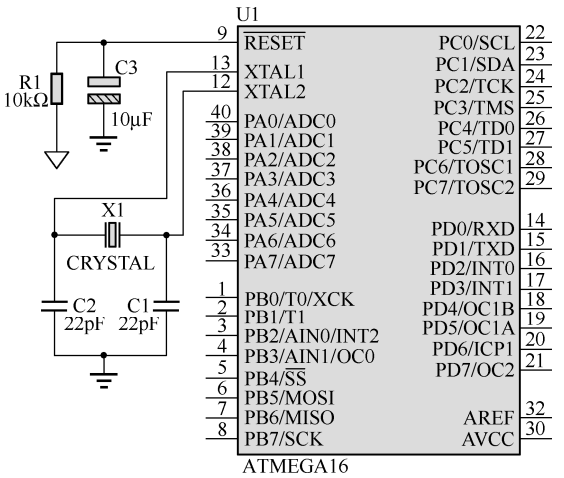


图 1-3　ATmega16 单片机最小系统电路原理图

图 1-3 所示的单片机最小系统通上电后，单片机即可开始工作，4 组端口的数据寄存器 PORTx 处于 0x00 状态；数据方向寄存器 DDRx 处于 0x00（表示为输入）状态；输入引脚寄存器 PINx 处于高阻状态。在这种情况下，是否说明单片机正常工作呢？单片机要完成相应的任务操作，还必须用程序来进行控制，没有固化程序的单片机系统是无法完成任何实质上的工作的。所以图 1-3 所示的电路，严格地说，通电后单片机进入了工作准备就绪状态。

2．时钟系统控制

【时钟系统和时钟分配】ATmega16 的主要时钟系统及其分布如图 1-4 所示，这些时钟并不需要同时工作。为了降低功耗，可以使用不同的睡眠模式来禁止无须工作的模块时钟。

ATmega16 时钟系统将产生的时钟信号主要有 clk_{CPU}（CPU 时钟）、$clk_{I/O}$（I/O 时钟）、clk_{Flash}（Flash 时钟）、clk_{ASY}（异步定时器时钟）、clk_{ADC}（ADC 时钟）。

clk_{CPU}（CPU 时钟）与操作 AVR 内核的子系统相连，如通用寄存器文件、状态寄存器及保存堆栈指针的数据存储器。如果终止 CPU 时钟，将使 CPU 内核停止工作。

$clk_{I/O}$（I/O 时钟）用于主要的 I/O 模块，如定时器/计数器、SPI 和 USART。$clk_{I/O}$ 还用于外部中断模块。由于某些外部中断是通过异步逻辑检测的，因此即使 $clk_{I/O}$ 停止了，这些中断仍然可以得到监控。此外，TWI 模块的地址识别功能在没有 $clk_{I/O}$ 的情况下也是异步实现的，使得这个功能在任何睡眠模式下都可以正常工作。

clk_{Flash} 控制 Flash 接口的操作，此时钟通常与 CPU 时钟同时挂起或激活。

clk_{ASY} 允许异步定时器/计数器与 LCD 控制器直接由外部 32kHz 时钟晶体驱动，使得此定时器/计数器即使在睡眠模式下仍然可以为系统提供一个实时时钟。

ADC 具有专门的时钟 clk_{ADC}。这样可以在 ADC 工作时，停止 CPU 和 I/O 时钟以降低数字电路产生的噪声，从而提高 ADC 转换精度。

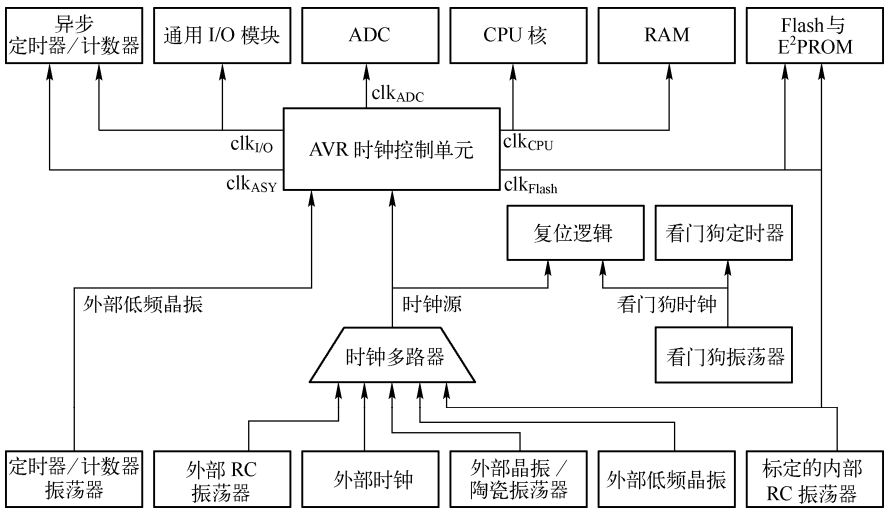


图 1-4　ATmega16 的主要时钟系统及其分布

【时钟源】ATmega16 芯片可以通过 Flash 熔丝位来选择系统的时钟源，见表 1-7。对于所有的熔丝位，“1”表示未编程；“0”表示已编程。时钟输入到 AVR 时钟发生器，再分配到相应的模块。器件出厂时，默认设置为 CKSEL=“0010”，SUT=“10”。这种默认设置的时钟源为 1MHz 的内部 RC 振荡器，启动时间最长，保证用户可以通过 ISP 或并行编程器得到所需的时钟源。

表 1-7　ATmega16 芯片时钟源选择

时 钟 系 统	CKSEL[3..0]	时 钟 系 统	CKSEL[3..0]
外部晶振/陶瓷振荡器	1111～1010	标定的内部 RC 振荡器	0100～0001
外部低频晶振	1001	外部时钟	0000
外部 RC 振荡器	1000～0101		

【外部晶体振荡器】XTAL1 与 XTAL2 分别为片内振荡器的反向放大器的输入端和输出端。在 XTAL1 与 XTAL2 引脚上外接振荡器和电容可组成谐振回路，配合片内的 OSC 振荡回路构成稳定的自激振荡器，如图 1-5 所示。这个外接振荡器可以使用石英晶体，也可以使用陶瓷谐振器。熔丝位 CKOPT 用于选择这两种放大器模式的其中之一。当 CKOPT 被编程时，振荡器在输出引脚产生满幅度的振荡。这种模式适合于噪声环境，以及需要通过 XTAL2 驱动第 2 个时钟缓冲器的情况，而且这种模式的频率范围比较宽。当保持 CKOPT 为未编程状态时，振荡器的输出信号幅度比较小，其优点是大大降低了功耗，但是频率范围比较窄，而且不能驱动其他时钟缓冲器。

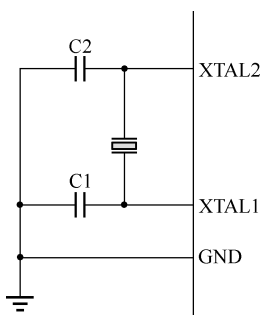


图 1-5　外部晶振连接图

对于谐振器，其频率可在 0～16MHz 之间选择，CKOPT 未编程时的选择谐振器的最高频率为 8MHz，CKOPT 编程时选择谐振器的最高频率为 16MHz。C1 和 C2 的数值要一样，不管使用的是晶体还是谐振器。最佳的数值不仅与使用的晶体或谐振器有关，还与杂散电容和环境电磁噪声有关。振荡器可以工作在 3 种不同的模式，每一种都有一个优化的频率范围，见表 1-8。

表 1-8　晶体振荡器工作模式

CKOPT	CKSEL[3..1]	频率范围/MHz	使用晶体时 C1 和 C2 电容的推荐值/pF
1	101 注	0.4～0.9	—
1	110	0.9～3.0	12～22
1	111	3.0～8.0	12～22
0	101、110、111	≥1.0	12～22

注：此选项只能用于陶瓷谐振器，不适用于晶体。

表 1-9 列出了熔丝位 CKSEL[0]及 SUT[1..0]用于选择的启动时间。

表 1-9　晶体振荡器的启动时间的选择

CKSEL[0]	SUT[1..0]	掉电与省电模式下的启动时间	复位时额外的延迟（VCC 为 5.0V）	推 荐 用 法
0	00	258CK 注1	4.1ms	陶瓷谐振器，电源快速上升
0	01	258CK 注1	65ms	陶瓷谐振器，电源缓慢上升
0	10	1KCK 注2	—	陶瓷谐振器，BOD 使能
0	11	1KCK 注2	4.1ms	陶瓷谐振器，电源快速上升
1	00	1KCK 注2	65ms	陶瓷谐振器，电源缓慢上升
1	01	16KCK	—	晶体振荡器，BOD 使能
1	10	16KCK	4.1ms	晶体振荡器，电源快速上升
1	11	16KCK	65ms	晶体振荡器，电源缓慢上升

说明：

注 1：这些选项不适用于晶体，只适用于陶瓷谐振器。只能用于工作频率不太接近于最大频率，而且启动时的频率稳定性对于应用而言不是十分重要的情况。

注 2：这些选项是为陶瓷谐振器设计的，可以保证启动时频率足够稳定。如果工作频率不太接近于最大频率，而且启动时的频率稳定性对于应用而言不十分重要时也适用于晶体。

【外部低频晶体振荡器】为了使用 32.768kHz 时钟晶体作为器件的时钟源，必须将熔丝位 CKSEL 设置为“1001”以选择低频晶体振荡器。外部低频晶体振荡的连接也采用如图 1-6 所示的方式。通过对熔丝位 CKOPT 的编程，用户可以使能 XTAL1 和 XTAL2 的内部电容（内部电容的标定数值为 36pF），从而去除外部电容。选择这个振荡器后，启动时间由熔丝位 SUT 确定，见表 1-10。

表 1-10　晶体振荡器工作模式

SUT[1..0]	掉电与省电模式下的启动时间	复位时额外的延迟（VCC 为 5.0V）	推 荐 用 法
00	1KCK 注	4.1ms	电源快速上升或 BOD 使能
01	1KCK 注	65ms	电源缓慢上升
10	32KCK	65ms	启动时频率已经确定
11	保留		

注：这些选项只能用于启动时频率稳定性对应用而言不十分重要的情况。

【外部 RC 振荡器】对于时间不敏感的应用可以使用图 1-6 所示的外部 RC 振荡器。频率可以通过方程 $f=1/(3RC)$进行粗略的估计。电容 C 至少要 22pF。通过编程熔丝位 CKOPT，用户可以使能 XTAL1 和 GND 之间的片内 36pF 电容，从而省去外部电容。

振荡器可以工作于 4 种不同的模式，每个模式有自己的优化频率。工作模式通过熔丝位 CKSEL[3..0]选取（见表 1-11）。

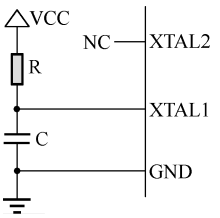


图 1-6　外部 RC 振荡器连接图

表 1-11　外部 RC 振荡器工作模式

CKSEL[3..0]	频率范围/MHz
0101	0～0.9
0110	0.9～3.0
0111	3.0～8.0
1000	8.0～12.0

选择了这个振荡器后，启动时间由熔丝位 SUT 确定，见表 1-12。

> 注意:
> 在 SUT[1..0]为“11”时，只能用于工作频率不太接近于最大频率时的情况。

表 1-12　外部 RC 振荡器的启动时间

SUT[1..0]	掉电与省电模式下的启动时间	复位时额外的延迟（VCC 为 5.0V）	推 荐 用 法
00	18KCK	—	BOD 使能
01	18KCK	4.1ms	电源快速上升
10	18KCK	65ms	电源缓慢上升
11	6KCK	4.1ms	电源快速上升或是 BOD 使能

【标定的片内 RC 振荡器】标定的片内 RC 振荡器提供了固定的 1.0MHz、2.0MHz、4.0MHz 或 8.0MHz 的时钟。这些频率都是在 5V、25℃条件下的标定值。这个时钟也可以作为系统时钟，只要按照表 1-13 所列对熔丝位 CKSEL 进行编程即可。选择这个时钟（此时不能对 CKOPT 进行编程）后就无须外部器件了。复位时，硬件将标定字节加载到 OSCCAL 寄存器，自动完成对 RC 振荡器的标定。在 5V、25℃和频率为 1.0MHz 时，这种标定可以提供标称频率的±1%的精度。当使用这个振荡器作为系统时钟时，看门狗仍然使用自己的看门狗定时器作为溢出复位的依据。

表 1-13　内部 RC 振荡器的熔丝位编程值

CKSEL[3..0]	标称频率/MHz	CKSEL[3..0]	标称频率/MHz
00001[注]	1.0	0011	4.0
0010	2.0	0100	8.0

注：出厂时的设置

选择这个振荡器后，启动时间由熔丝位 SUT 确定，见表 1-14。XTAL1 和 XTAL2 要保持为空。

表 1-14　内部标定 RC 振荡器的启动时间

SUT[1..0]	掉电与省电模式下的启动时间	复位时额外的延迟（VCC 为 5.0V）	推 荐 用 法
00	6CK	4.1ms	BOD 使能
01	6CK	65ms	电源快速上升
10[注]	6CK	65ms	电源缓慢上升
11	保留		

注：出厂时的设置

振荡器标定寄存器 OSCCAL 定义如下：

Bit	7	6	5	4	3	2	1	0
	CAL7	CAL6	CAL5	CAL4	CAL3	CAL2	CAL1	CAL0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	标定数值							

☺ CAL7 ~ CAL0（Bit7 ~ Bit0）为振荡器标定数值。将标定数值写入这个地址可以对内部振荡器进行调节，以消除由于生产工艺所带来的振荡器频率偏差。复位时，1MHz 的标定数据（标志数据的高字节，地址为 0x00）自动加载到 OSCCAL 寄存器。如果需要内部 RC 振荡器工作于其他频率，标定数据必须人工加载，即首先通过编程器读取标志数据，然后将标定数据保存到 Flash 或 E^2PROM 中。这些数据可以通过软件读取，然后加载到 OSCCAL 寄存器。当 OSCCAL 为“0x00”时，振荡器以最低频率工作；当对其写不为 0 的数据时，内部振荡器的频率将增加；写入 0xFF 即得到最高频率。标定的振荡器用于为访问 E^2PROM 和 Flash 定时。在写 E^2PROM 和 Flash 的操作时，不要将频率标定到超过标称频率的 10%，否则写操作有可能失败。

📖 注意：

振荡器只对 1.0MHz、2.0MHz、4.0MHz 和 8.0MHz 这 4 种频率进行了标定，其他频率则无法保证，见表 1-15。

表 1-15　内部 RC 振荡器频率范围

OSCCAL 数值	最小频率与标称频率的百分比/%	最大频率与标称频率的百分比/%
0x00	50	100
0x7F	75	150
0xFF	100	200

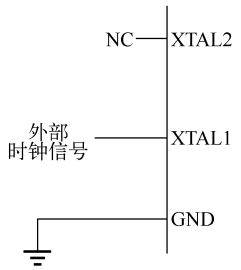


图 1-7　外部时钟配置图

【外部时钟源】为了从外部时钟源驱动芯片，XTAL1 必须按图 1-7 所示的方法进行连接。同时，熔丝位 CKSEL 必须编程为“0000”。若熔丝位 CKOPT 也被编程，用户就可以使用内部的 XTAL1 和 GND 之间的 36pF 电容。选择了这个振荡器后，启动时间由熔丝位 SUT 确定，见表 1-16。

为了保证 MCU 能够稳定工作，不能突然改变外部时钟源的振荡频率。工作频率突变超过 2%将会产生异常现象。应该在 MCU 保持复位状态时改变外部时钟的振荡频率。

表 1-16　外部时钟的启动时间

SUT[1..0]	掉电与省电模式下的启动时间	复位时额外的延迟（VCC 为 5.0V）	推 荐 用 法
00	6CK	—	BOD 使能
01	6CK	4.1ms	电源快速上升
10	6CK	65ms	电源缓慢上升
11	保留		

【定时器/计数器振荡器】对于拥有定时器/振荡器引脚（TOSC1 和 TOSC2）的 AVR 微处理器，晶体可以直接与这两个引脚连接，无须外部电容。此振荡器针对 32.768kHz 的时钟晶体作了优化。建议不在 TOSC1 引脚输入振荡信号。

3．电源管理——睡眠模式

睡眠模式可以使应用程序关闭 MCU 中没有使用的模块，从而降低功耗。AVR 具有不同的睡眠模式，允许用户根据自己的应用要求实施剪裁。进入睡眠模式的条件是置位寄存器 MCUCR 的 SE，然后执行 SLEEP 指令。具体哪一种模式（空闲模式、ADC 噪声抑制模式、掉电模式、省电模式、Standby 模式或扩展 Standby 模式）由 MCUCR 的 SM2、SM1 和 SM0 决定，见表 1-17。使能的中断可以将进入睡眠模式的 MCU 唤醒。经过启动时间，外加 4 个时钟周期后，MCU 就可以运行中断服务程序了，然后返回到 SLEEP 的下一条指令。唤醒时不会改变寄存器文件和 SRAM 的内容。如果在睡眠过程中发生了复位，则 MCU 唤醒后从中断向量处开始执行。

表 1-17　休眠模式选择

SM2	SM1	SM0	休 眠 模 式	SM2	SM1	SM0	休 眠 模 式
0	0	0	空闲模式	1	0	0	保留
0	0	1	ADC 噪声抑制模式	1	0	1	保留
0	1	0	掉电模式	1	1	0	Standby 模式[注]
0	1	1	省电模式	1	1	1	扩展 Standby 模式[注]

注：仅在使用外部晶振或谐振器时 Standby 模式下才能使用。

ATmega16 控制寄存器 MCUCR 包含了电源管理的控制位，各位的定义如下：

Bit	7	6	5	4	3	2	1	0
	SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ SM2 ~ SM0（Bit7、Bit5、Bit4）：休眠模式选择位。

☺ SE（Bit6）：休眠使能。为了使 MCU 在执行 SLEEP 指令后进入休眠模式，SE 必须置“1”。为了确保进入休眠模式，建议在 SLEEP 指令的前一条指令将 SE 置“1”。MCU 一旦唤醒立即清除 SE 位。

4．复位系统控制

ATmega16 有 5 个复位源，即上电复位、外部复位、看门狗复位、掉电检测复位和 JTAG 复位。复位时，所有的 I/O 寄存器都被设置为初始值，程序从复位向量处开始执行。所有的复位信号消失后，芯片内部的 1 个延迟计数器被激活，将内部复位的时间延长。这种处理方式使得在 MCU 正常工作前有一定的时间让电源达到稳定的电平。

MCUCSR 为 MCU 控制和状态寄存器，它提供了有关引起 MCU 复位的复位源的信息，各位的定义如下：

Bit	7	6	5	4	3	2	1	0
	JID	ISC2	—	JIRF	WDRF	BORF	EXTRF	PORF
读/写	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ JTD（Bit7）：禁止/允许 JTAG 接口。此位为“0”时，如果 JTAGEN 熔丝位被编程，则 JTAG 接口使能。如果该位为“1”，JTAG 接口禁止。

☺ ISC2（Bit6）：中断 2 触发方式控制。

☺ JTRF（Bit4）：JTAG 复位标志。通过 JTAG 指令 AVR_RESET 可以使 JTAG 复位寄存器置位，并引发 MCU 复位，并使 JTRF 置“1”。上电复位将使其清“0”，也可以通过软件清“0”。

☺ WDRF（Bit3）：看门狗复位标志。看门狗复位发生时置“1”。上电复位将使其清“0”，也可以通过软件清“0”。

☺ BORF（Bit2）：掉电检测复位标志。掉电检测复位发生时置“1”。上电复位将使其清“0”，也可以通过软件清“0”。

☺ EXTRF（Bit1）：外部复位标志。外部复位发生时置“1”。上电复位将使其清“0”，也可以通过软件清“0”。

☺ PORF（Bit0）：上电复位标志。上电复位发生时置“1”，只能通过软件清“0”。

任务 4　自制下载线

由于 AVR 单片机采用了 Flash 存储器结构，如果对 AVR 单片机的 MOSI、RESET、SCK、MISO 引脚提供相关的信号，则可以将.hex 或.coff 文件和熔丝位的配置固化到 AVR 单片机中，从而实现 ISP（In System Programmer）在系统可编程功能。根据使用计算机端口的类型不同，单片机 ISP 下载线可分为 USB 下载线、串行 ISP 下载线、并行 ISP 下载线

等。制作 USB 下载线时，在该下载线电路中需使用 MCU，且需对此 MCU 安装 USB 驱动程序，因此对于一般读者来说，可能有一定的难度。串行 ISP 下载线和并行 ISP 下载线制作起来相对简单，所以在此介绍这两种类型的下载线的制作。

1．串行 ISP 下载线的制作

串行 ISP 下载线的原理是使用计算机的串行口（COM）来驱动 ISP 编程所需的信号波形，从而将已编译生成的.hex 或.coff 文件和熔丝位的配置固化到单片机中，以实现在系统可编程。串行 ISP 下载线的电路原理图如图 1-8 所示，图中的 DB9 与计算机 COM 口连接；MAX232 用于计算机与 AVR 单片机之间的电平转换，以解决二者间电平不兼容的问题，从而使计算机的串行口不易损坏；J1 为 ISP 下载线的接口，通过此接口与单片机的相关引脚相连接。

> 注意：
> 此下载线使用了 AVR 单片机目标板上的电源。

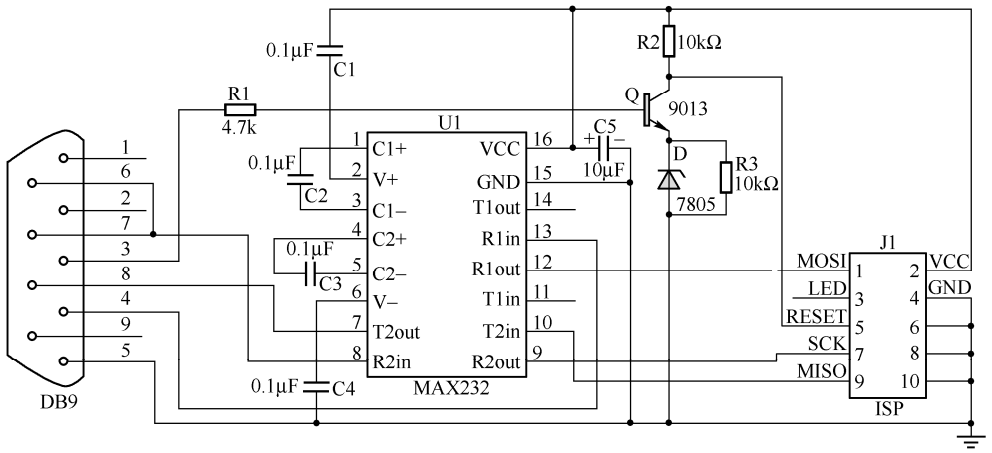


图 1-8　串行下载线原理图

2．并行 ISP 下载线的制作

并行 ISP 下载线的原理是使用计算机的并行口（LPT 打印机接口）来驱动 ISP 编程所需的信号波形，从而将已编译生成的.hex 或.coff 文件和熔丝位的配置固化到单片机中，以实现在系统可编程。并行 AVR ISP 下载线的优点是价格便宜、结构简单，它里面就是一个 74HC244 芯片和数个阻容元件，缺点是体积大，需要占用一个打印机的并行接口。并行 ISP 下载线的电路原理图如图 1-9 所示，图中的 DB25 与计算机的打印机接口相连；高速缓冲器 74HC244 用于保护计算机打印机并行口，避免在使用中误操作而损坏打印机并行口；J1 为 ISP 下载线的接口，通过此接口与单片机的相关引脚相连接。

> 注意：
> 此下载线使用了 AVR 单片机目标板上的电源。

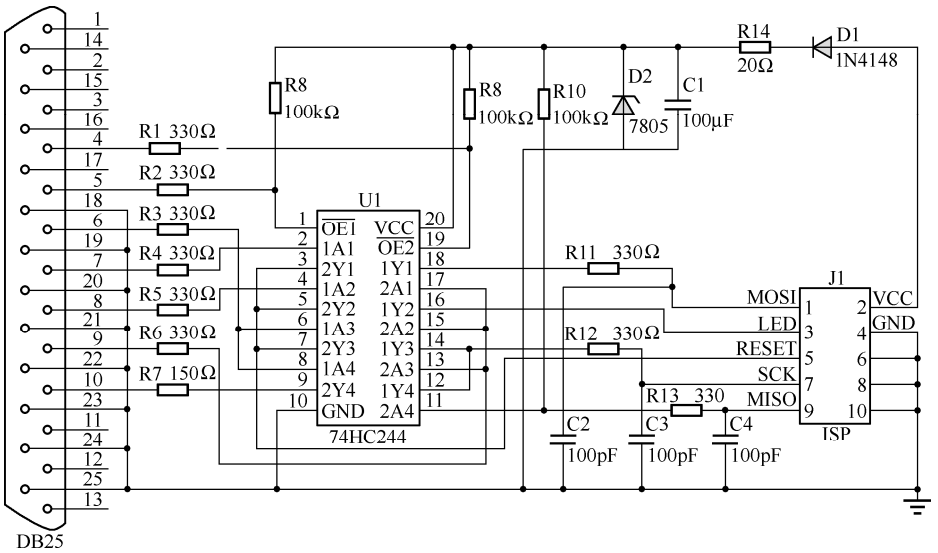


图 1-9　并行下载线原理图

项目二　单片机系统开发软件的应用

〖知识目标〗
☺ 单片机编译软件的使用方法;
☺ 单片机程序固化软件的使用方法;
☺ 单片机虚拟仿真软件的使用方法。
〖能力目标〗
☺ 会利用 CVAVR 软件对单片机 C 语言程序进行编译;
☺ 会利用下载软件进行程序固化;
☺ 会利用 Proteus7.10 软件进行单片机程序的硬件仿真。

任务 1　CVAVR 编译软件的使用

CodeVision AVR C Compiler（简称 CVAVR 编译器）是为 Atmel AVR 系列微控制器而设计的一款 C 编译器, 可以在 Windows95/98/NT 4.0/2000/XP/Vista 操作系统下运行。

一个 CVAVR 项目就是一组文件和编译器设置的集合，它可以用于建立一个具体的程序，项目文件的扩展名为.prj。要建立一个完整的项目，并将源代码生成可执行代码下到目标芯片，可通过以下步骤进行。

1．新建项目

执行菜单命令“开始”→“程序”→“CodeVisionAVR”→“CodeVisionAVR C Compiler”，或者在桌面上双击图标，进入 CVAVR 集成工作环境界面（IDE）。

在 IDE 中执行菜单命令“File”→“New”，或者在工具栏中单击图标，将弹出如图 2-1 所示的“Create New File”对话框。在此对话框中选择“Source”时将会创建一个.c 源程序文件，选择“Project”时将会创建一个项目文件。

在图 2-1 中选择“Project”，并单击“OK”按钮时，将弹出如图 2-2 所示的“Confirm”对话框。在此对话框中，要求用户确认是否使用 CVAVR 新建项目，若单击“Yes”按钮，则会打开“CVAVR”对话框，通过 CVAVR 项目向导来创建项目；若单击“No”按钮，将弹出如图 2-3 所示的“Create New Project”对话框。

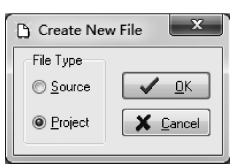


图 2-1　创建新文件对话框



图 2-2　CodeVisionAVR 的“Confirm”对话框

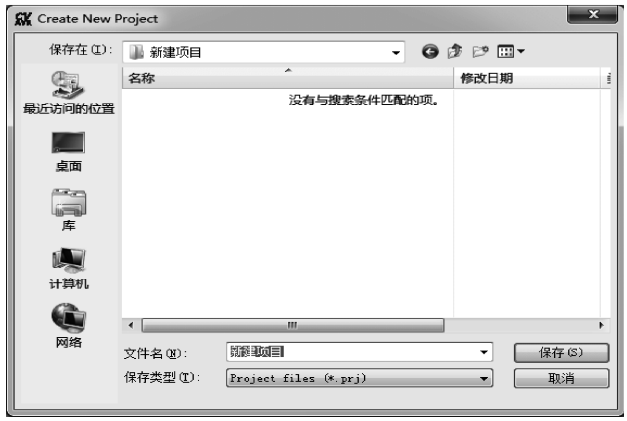


图 2-3　创建新项目对话框

在图 2-3 中选择合适的保存路径并输入项目名，然后单击“保存”按钮，将弹出 Configure 配置文件对话框。在此对话框的 C Compiler 选项卡中设置“Chip”栏为“ATmega16”，“Clock”栏为 8MHz，如图 2-4 所示。

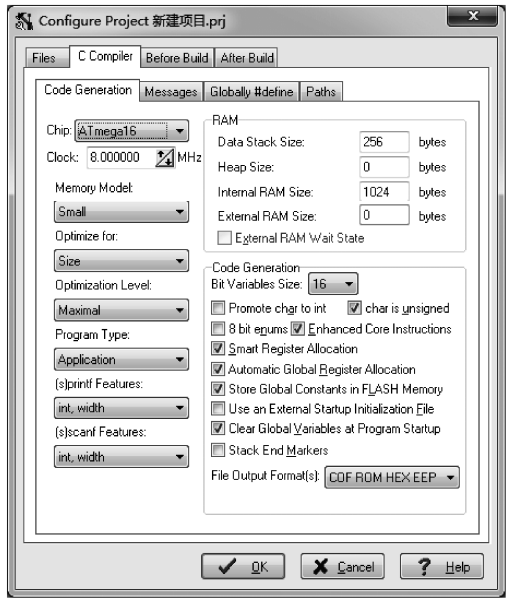


图 2-4　Configure 的 C Compiler 对话框

在 Configure 配置文件对话框中进行相关设置后，若单击“OK”按钮，将会回到 CVAVR 的 IDE 界面，只不过此时 IDE 中将会新增一个“Project Notes”窗口，用户可以在此窗口中添加相关的项目说明信息，如图 2-5 所示。至此已经简单地创建了一个新项目，此项目名为“新建项目.prj”。

2．创建源程序文件

创建好项目后，在 IDE 中执行菜单命令“File”→“New”，或者在工具栏中单击图标，在弹出的“Create New File”对话框中选择“Source”。此时，在 IDE 中会为新建的文件

打开一个新的编辑器窗口，其标题栏显示文件名为 untitled.c，该文件列在文件导航器的 Other Files 目录下。

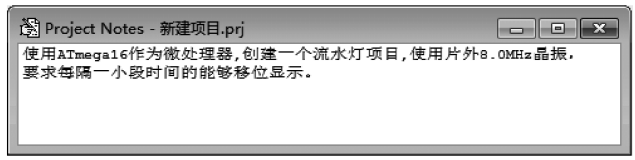


图 2-5　Project Notes 窗口

在刚创建的编辑器窗口中输入如下源程序文件，并执行菜单命令“File”→“Save”，或者在工具栏中单击图标，然后选择合适的保存路径，且输入源程序的文件名（假设此文件名为“新建一个项目.c”）将其进行保存，此时文件导航器的 Other Files 目录下变为“新建一个项目.c”。

```
/*****************************************************
 File name:              新建一个项目.c
 Chip type:              ATmega16
 Clock frequency:        8.0MHz
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
flash unsigned char tab[]={0xfe,0xfd,0xfb,0xf7,0xef,0xdf,0xbf,0x7f};
void delay(uint k)
{    uint m,n;
      for(m=0;m<k;m++)
            {
              for(n=0;n<1000;n++);
            }
}
void main(void)
{
 char position;
   DDRC=0xFF;
    while(1)
     {
       for(position=0;position<8;position++)
          {
            PORTC=tab[position];
            delay(1000);
          }
     }
}
```

3．将源程序文件添加到项目

在 IED 中执行菜单命令“Project”→“Configure”，或者在工具栏中单击图标，将弹

出“Configure Project”对话框，在此对话框的“File”栏中单击“Add”按钮，然后依据提示将刚才所创建的源程序文件添加到项目中，此时文件导航器 Other Files 目录的“新建一个项目.c”已经转移到“Project:新建项目”下，表示“新建一个项目.c”已经是所创建项目中的一部分了，如图 2-6 所示。

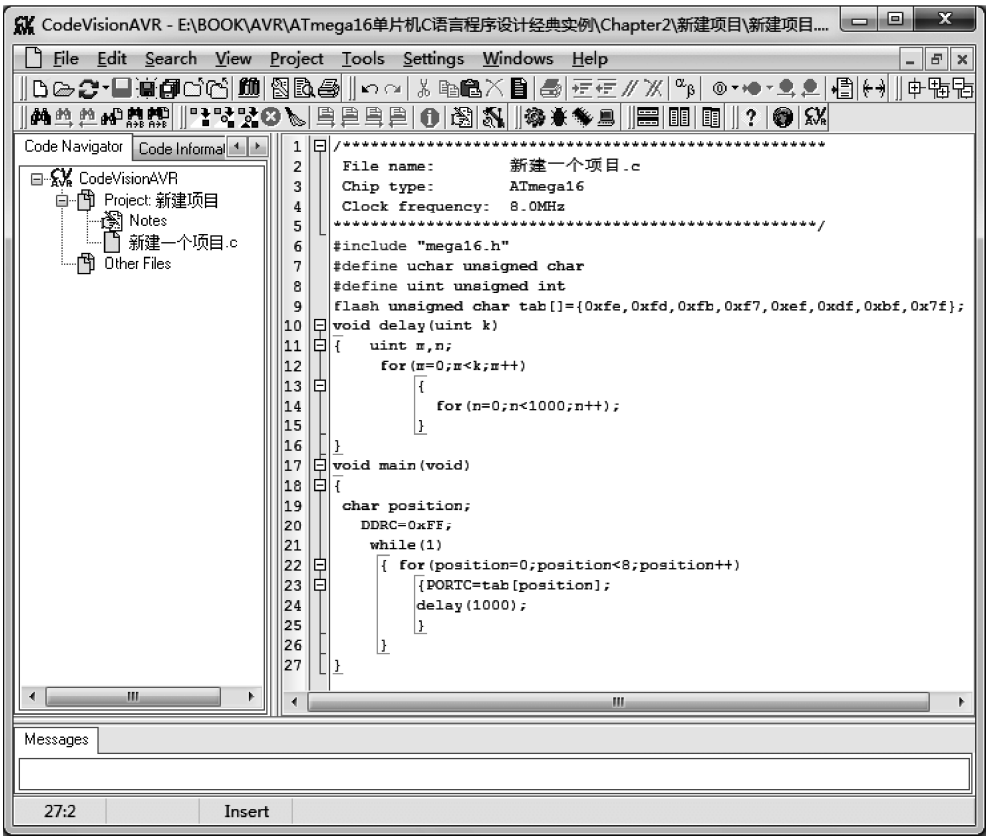


图 2-6　添加“新建一个项目.c”到项目中

4．在 CVAVR 中编译并生成项目

源程序编写完，并将其添加到项目中后，就可对源程序进行编译。在 IDE 中执行菜单命令“Project”→“Compile”，或者在工具栏中单击图标，会弹出一个“Information”对话框显示编译结果，同时在 LIST 文件夹中产生一个“新建项目.asm”的汇编程序源文件。“Information”对话框的顶部列出了编译器的选项，下面是被编译的代码行数和警告、错误个数。对话框的其余部分列出了存储器的使用和用途。

如果在编译过程中有错误或警告产生，它们就会出现在编译器窗口下面的消息窗口（Message）和导航器窗口中。双击错误或警告信息可以高亮显示出问题的代码行。

生成项目会创建一个可执行文件。在 IDE 中执行菜单命令“Project”→“Build”（或“Build All”），或者在工具栏中单击图标（或），也会弹出一个“Information”对话框显示编译结果并产生一个“新建项目.asm”的汇编程序源文件。只不过此时“Information”对话框中有两个选项卡，即“Complier”和“Assembler”。“Assembler”选项卡顶部显示了编译的版本、编译的步骤及存储器的使用情况。

注意：

Compile 命令和 Build（或 Build All）命令是不同的：Compile 命令仅进行源代码的编译，但不产生二进制目标文件，而 Build（或 Build All）命令能产生相应的二进制代码文件。

5．导出到 AVR Studio 进行项目调试

源程序通过编译，只能说明程序的语法正确，并不代表所编写的程序已达到了设计要求。那么怎样判断程序是否正确呢？可以通过实验的方法来检验。最直接的方法就是将生成的二进制代码（如.HEX 或.COF 文件）下载到目标芯片中运行，通过实际的运行来检验是否达到设计要求。这种方法虽然最正确，但是不能告诉用户程序错误是什么引起的，尤其对于经验不足的用户来说，更不容易找出问题所在。另外一种方法是使用仿真或模拟的方法来进行程序的调试。

CVAVR 不提供在线调试、仿真功能，因此需要第三方调试工具来进行程序的调试。通常硬件仿真的价格都比较高，所以在很多时候可以通过软件模块器来进行程序的调试，如 AVR Studio 和 Proteus 等。

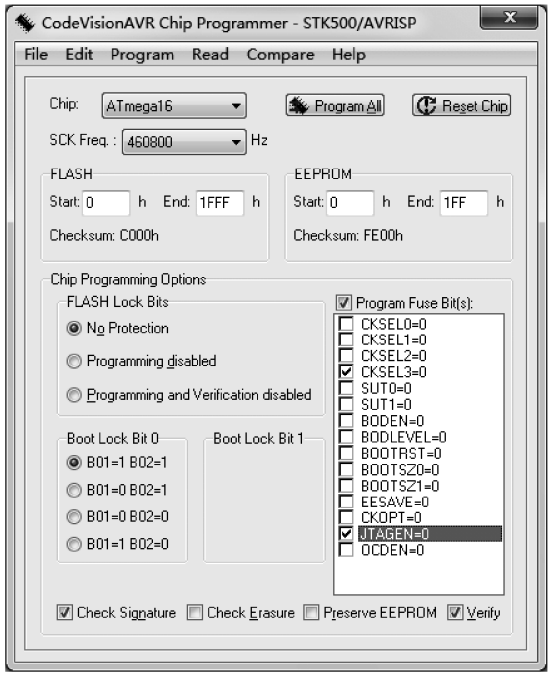


图 2-7　Chip Programmer 界面

编译好程序后，执行菜单命令“Tools”→“Debugger”，或者单击工具栏上的图标，即可进入 AVR Studio 进行程序调试。其具体的调试方法将在本项目的任务 2 中进行讲述。

6．对目标设备芯片编程

CodeVisionAVR IDE 集成支持串口/并口 AVRISP 的下载烧写软件。用户将编程器通过串行口或打印机并行端口与计算机连接后，在 CodeVisionAVR IDE 中可直接将编译调试好的源程序固化到目标芯片中。

在固化程序前，执行菜单命令“Setting”→“Programmer”，在弹出的对话框中选择合适的编程器类型及所使用的计算机端口。

在 CodeVisionAVR IDE 中执行菜单命令“Tools”→“Chip Programmer”，或者在工具栏中单击图标，将弹出如图 2-7 所示的“CodeVisionAVR Chip Programmer-STK500/AVRISP”对话框。它主要包括 Chip（芯片）、Flash 和 E^2PROM、Flash Lock Bits（Flash 存储器锁存位）、Program Fuse Bit（可编程熔丝位）等内容的设置。

【芯片】“Chip”栏中的选项决定了“CodeVisionAVR Chip Programmer-STK500/AVRISP”对话框中其他可用的选项，在该栏中可以选择合适的芯片类型。芯片的类型不同，使得 Flash 和 E^2PROM 的缓存容量大小也不相同，并且也影响了“Program Fuse Bit(s)”区域中相关位的设定和 Boot Lock Bit0、Boot Lock Bit1 的设定。

【Flash 和 E^2PROM】编程器有两个存储器缓冲区，即 Flash 存储器缓冲区和 E^2PROM 存储器缓冲区。缓冲区的内容可以通过执行菜单命令“File”→“Load Flash”（或“Load EEPROM”）进行加载，或者执行菜单命令“File”→“Save Flash”（或“Save EEPROM”）进行保存。

在一个文件被加载到相应的缓冲区后，其起始（Start）地址和结束（End）地址会相应更新，同时 Checksum 的值也会发生相应的更改。

执行菜单命令“Edit”→“FLASH”，将弹出如图 2-8 所示的“Edit FLASH Buffer”对话框，在此可以查看和修改 Flash 存储器缓冲区中的内容。执行菜单命令“Edit”→“EEPROM”，在弹出的对话框中可以查看和修改 E^2PROM 存储器缓冲区中的内容。

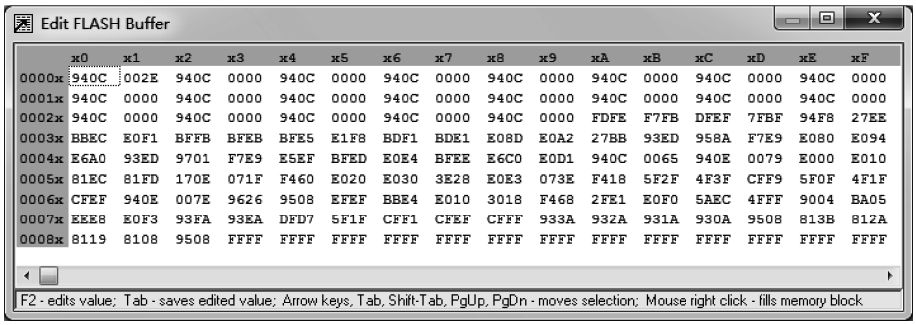


图 2-8　Flash 存储器缓冲区

【Flash 存储器锁存位（Flash Lock Bits）】“Chip Programmer Options”区域中的“FLASH Lock Bits”选项组有 3 个选项可能选择芯片编程使用的安全级别，其中“No Protection”表示允许所有的读取的编程操作；“Programming disabled”表示对 Flash 和 E^2PROM 存储区只能读不能写；“Programming and Verification disabled”表示不能对芯片内容进行任何读/写操作。

执行菜单命令“Program”→“All”，锁定位将自动写入目标芯片中；执行菜单命令“Program”→“Lock Bit”，可以单独写入锁定位；执行菜单命令“Read”→“Lock Bit”，可以读取锁定位当前值；执行菜单命令“Program”→“Erase Chip”，可以将所有的锁定位设为“No Protection”。

【可编程熔丝位（Program Fuse Bit）】“Chip Programmer Options”区域中的“Program Fuse Bit(s)”选项组中有各种复选框，每个对应目标芯片上的一个熔丝位。

CKSEL 和 SUT 熔丝位决定 AVR 单片机所采用的时钟及复位启动延时时间，其设置含义见表 2-1。BODEN（Brown Out Detection Enable）为掉电检测电路的功能控制，BODEN=1 时，禁止掉电检测；BODEN=0 时，允许掉电检测。BODLEVEL 为掉电检测电路的电平选择控制，BODLEVEL=1 时，选择 2.7V 电平；BODLEVEL=0 时，选择 4.0V 电平。BOOTRST 为复位入口地址选择，BOOTRST=1 时，程序从 0x0000 地址开始；BOOTRST=0 时，复位后从 BOOT 区执行。BOOTSZ（BOOTSZ1、BOOTSZ0）可设置引导区程序大小及入口，BOOTSZ=00，引导区为 1024 字，入口为 0xc00；BOOTSZ=01，引导区为 512 字，入口为 0xe00；BOOTSZ=10，引导区为 256 字，入口为 0xf00；BOOTSZ=11，引导区为 128 字，入口为 0xf80。EEAVE 可设置程序固化时 E^2PROM 中的数据是否保留，EEAVE=1，不保留；EEAVE=0，保留。CKOPT 可用于选择放大器模式，CKOPT=0，高幅度振荡输出；CKOPT=1，低幅度振荡输出。JTAGEN 设置 JTAG 测试访问端口状态，OCDEN（On Chip Debug Enable）片上调试使能位。在进行 JTAG 调试时，应将 JTAGEN 和 OCDEN 这两位都选中，并保持所有的锁定位处于非锁定状态；在实际使用时为了降低功耗，不选中 JTAGEN 和 OCDEN 这两位，可以减少 2～3mA 的电流。Check Signature 可用于设置是否对该芯片进行签名。Check Erasure 用于设置是否检查芯片擦除的正确性。Preserve E^2PROM 可用于设置是否通过芯片擦除循环来保存 E^2PROM 的数据。Verify 可用于设置是否检查 Flash 和 E^2PROM 编程的正确性。

表 2-1　CKSEL 和 SUT 熔丝位的设置

CKSEL3	CKSEL2	CKSEL1	CKSEL0	SUT1	SUT0	设 置 说 明
0	0	0	0	0	0	外部时钟，启动延时为 6 CK+0ms
0	0	0	0	0	1	外部时钟，启动延时为 6 CK+4.1ms
0	0	0	0	1	0	外部时钟，启动延时为 6 CK+65ms
0	0	0	1	0	0	内部 RC 振荡 1MHz 时钟，启动延时为 6 CK+0ms
0	0	0	1	0	1	内部 RC 振荡 1MHz 时钟，启动延时为 6 CK+4.1ms
0	0	0	1	1	0	内部 RC 振荡 1MHz 时钟，启动延时为 6 CK+65ms
0	0	1	0	0	0	内部 RC 振荡 2MHz 时钟，启动延时为 6 CK+0ms
0	0	1	0	0	1	内部 RC 振荡 2MHz 时钟，启动延时为 6 CK+4.1ms
0	0	1	0	1	0	内部 RC 振荡 2MHz 时钟，启动延时为 6 CK+65ms
0	0	1	1	0	0	内部 RC 振荡 4MHz 时钟，启动延时为 6 CK+0ms
0	0	1	1	0	1	内部 RC 振荡 4MHz 时钟，启动延时为 6 CK+4.1ms
0	0	1	1	1	0	内部 RC 振荡 4MHz 时钟，启动延时为 6 CK+65ms
0	1	0	0	0	0	内部 RC 振荡 8MHz 时钟，启动延时为 6 CK+0ms
0	1	0	0	0	1	内部 RC 振荡 8MHz 时钟，启动延时为 6 CK+4.1ms
0	1	0	0	1	0	内部 RC 振荡 8MHz 时钟，启动延时为 6 CK+65ms
0	1	0	1	0	0	外部 RC 振荡≤0.9MHz，启动延时为 18 CK+0ms
0	1	0	1	0	1	外部 RC 振荡≤0.9MHz，启动延时为 18 CK+4.1ms
0	1	0	1	1	0	外部 RC 振荡≤0.9MHz，启动延时为 18 CK+65ms
0	1	0	1	1	1	外部 RC 振荡≤0.9MHz，启动延时为 6 CK+4.1ms
0	1	1	0	0	0	外部 RC 振荡 0.9～3MHz，启动延时为 18 CK+0ms
0	1	1	0	0	1	外部 RC 振荡 0.9～3MHz，启动延时为 18 CK+4.1ms

续表

CKSEL3	CKSEL2	CKSEL1	CKSEL0	SUT1	SUT0	设 置 说 明
0	1	1	0	1	0	外部 RC 振荡 0.9～3MHz，启动延时为 18 CK+65ms
0	1	1	0	1	1	外部 RC 振荡 0.9～3MHz，启动延时为 6 CK+4.1ms
0	1	1	1	0	0	外部 RC 振荡 3～8MHz，启动延时为 18 CK+0ms
0	1	1	1	0	1	外部 RC 振荡 3～8MHz，启动延时为 18 CK+4.1ms
0	1	1	1	1	0	外部 RC 振荡 3～8MHz，启动延时为 18 CK+65ms
0	1	1	1	1	1	外部 RC 振荡 3～8MHz，启动延时为 6 CK+4.1ms
1	0	0	0	0	0	外部 RC 振荡 8～12MHz，启动延时为 18 CK+0ms
1	0	0	0	0	1	外部 RC 振荡 8～12MHz，启动延时为 18 CK+4.1ms
1	0	0	0	1	0	外部 RC 振荡 8～12MHz，启动延时为 18 CK+65ms
1	0	0	0	1	1	外部 RC 振荡 8～12MHz，启动延时为 6 CK+4.1ms
1	0	0	1	0	0	低频晶振 32.768kHz，启动延时为 1K CK+4.1ms
1	0	0	1	0	1	低频晶振 32.768kHz，启动延时为 1K CK+65ms
1	0	0	1	1	0	低频晶振 32.768kHz，启动延时为 32K CK+65ms
1	0	1	0	0	0	低频石英/陶瓷振荡器 0.4～0.9MHz，258 CK+4.1ms
1	0	1	0	0	1	低频石英/陶瓷振荡器 0.4～0.9MHz，258 CK+65ms
1	0	1	0	1	0	低频石英/陶瓷振荡器 0.4～0.9MHz，1K CK+0ms
1	0	1	0	1	1	低频石英/陶瓷振荡器 0.4～0.9MHz，1K CK+4.1ms
1	0	1	1	0	0	低频石英/陶瓷振荡器 0.4～0.9MHz，1K CK+65ms
1	0	1	1	0	1	低频石英/陶瓷振荡器 0.4～0.9MHz，16K CK+0ms
1	0	1	1	1	0	低频石英/陶瓷振荡器 0.4～0.9MHz，16K CK+4.1ms
1	0	1	1	1	1	低频石英/陶瓷振荡器 0.4～0.9MHz，16K CK+65ms
1	1	0	0	0	0	中频石英/陶瓷振荡器 0.9～3MHz，258 CK+4.1ms
1	1	0	0	0	1	中频石英/陶瓷振荡器 0.9～3MHz，258 CK+65ms
1	1	0	0	1	0	中频石英/陶瓷振荡器 0.9～3MHz，1K CK+0ms
1	1	0	0	1	1	中频石英/陶瓷振荡器 0.9～3MHz，1K CK+4.1ms
1	1	0	1	0	0	中频石英/陶瓷振荡器 0.9～3MHz，1K CK+65ms
1	1	0	1	0	1	中频石英/陶瓷振荡器 0.9～3MHz，16K CK+0ms
1	1	0	1	1	0	中频石英/陶瓷振荡器 0.9～3MHz，16K CK+4.1ms
1	1	0	1	1	1	中频石英/陶瓷振荡器 0.9～3MHz，16K CK+65ms
1	1	1	0	0	0	高频石英/陶瓷振荡器 3～8MHz，258 CK+4.1ms
1	1	1	0	0	1	高频石英/陶瓷振荡器 3～8MHz，258 CK+65ms
1	1	1	0	1	0	高频石英/陶瓷振荡器 3～8MHz，1K CK+0ms
1	1	1	0	1	1	高频石英/陶瓷振荡器 3～8MHz，1K CK+4.1ms
1	1	1	1	0	0	高频石英/陶瓷振荡器 3～8MHz，1K CK+65ms
1	1	1	1	0	1	高频石英/陶瓷振荡器 3～8MHz，16K CK+0ms
1	1	1	1	1	0	高频石英/陶瓷振荡器 3～8MHz，16K CK+4.1ms
1	1	1	1	1	1	高频石英/陶瓷振荡器 3～8MHz，16K CK+65ms

在“Code Vision AVR Chip Programmer-STK500/AVRISP”对话框中执行菜单命令“Program”→“All”，可将源程序及设定的熔丝位的相关信息下载到目标芯片中。

任务 2　AVR Studio 集成开发软件的使用

AVR Studio 是 Atmel AVR 系列的软件仿真器，它由 Atmel 公司开发并免费提供给用户，用户可以在 Atmel 公司的官方网络（http://www.atmel.com）上自由下载。以本节 CVAVR 中已经创建的“新建项目”为例，讲述源程序在 AVR Studio 中的调试与仿真等内容。

1．将 CVAVR 中的源程序导入到 AVR Studio

在 CVAVR 中将源程序编译好后，若想将源程序导入到 AVR Studio 中，需要按下述步骤进行相关操作。

【步骤 1】在 CVAVR IDE 中，执行菜单命令“Tools”→“Debugger”，或者单击工具栏上的图标，将出现图 2-9 所示的对话框。

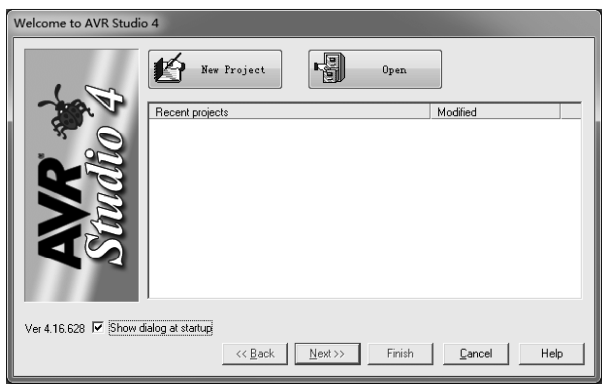


图 2-9　欢迎对话框

在图 2-9 所示的对话框中，若单击“New Project”按钮，将创建 Atmel AVR Assembler（汇编语言）或 AVR GCC（C/C++语言）项目。单击“Open”按钮，可打开已存在的项目。

【步骤 2】在图 2-9 所示对话框中单击“Open”按钮，弹出“Open Project or Object File”对话框，在“查找范围”栏中找到在 CVAVR 中已创建并需要打开的项目，如图 2-10 所示。

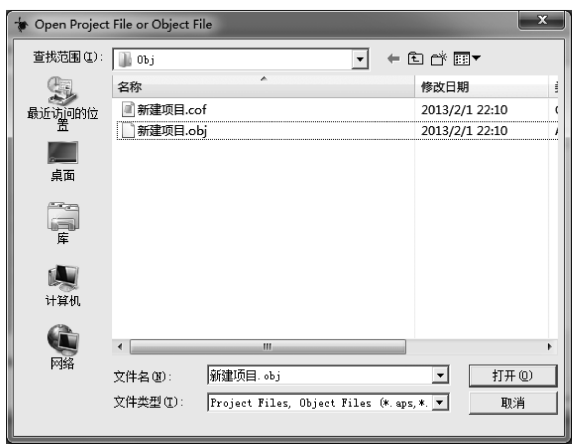


图 2-10　打开在 CVAVR 中已创建的项目

【步骤 3】找到项目后，单击“打开”按钮，弹出“Save AVR Studio Project File”对话框，询问用户在 AVR Studio 中保存项目的名称。用户可以更改项目名，也可直接单击“保存”按钮以使用默认的项目名。

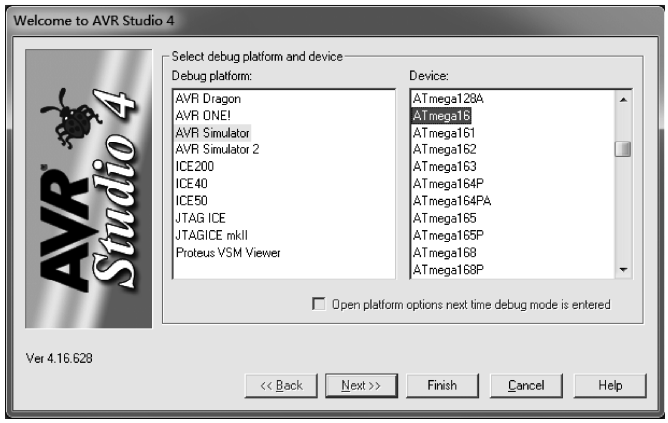


图 2-11　调试平台和器件选择对话框

【步骤 4】在“Save AVR Studio Project File”对话框中单击“保存”按钮，弹出图 2-11 所示的“Welcome to AVR Studio 4”对话框。在此对话框中选择“AVR Simulator”和“ATmega16”。这一步主要是为调试程序用的，也可以暂时不选，等到调度程序时，通过“Debug”菜单下的“AVR Simulator Options”命令来选择。选择完毕后，单击“Finish”按钮，将进入 AVR Studio 集成工作环境界面，如图 2-12 所示。

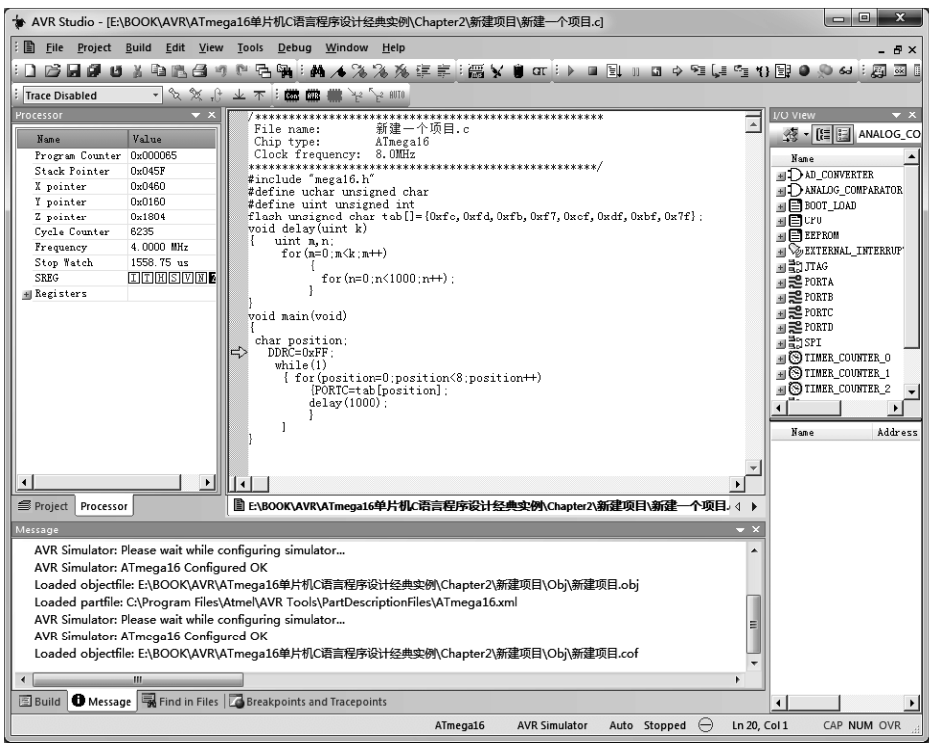


图 2-12　AVR Studio 集成工作环境界面

2．AVR Studio 仿真参数设置

在进行仿真前，还需要进行相关参数的设置，如更改 AVR 单片机型号、更改单片机的晶振频率等。在 AVR Studio 中执行菜单命令“Debug”→“AVR Simulator Options”，弹出“Simulator Options”对话框，如图 2-13 所示。“Simulator Options”对话框中有两个选项卡，即“Device selection”和“Stimuli and logging”。

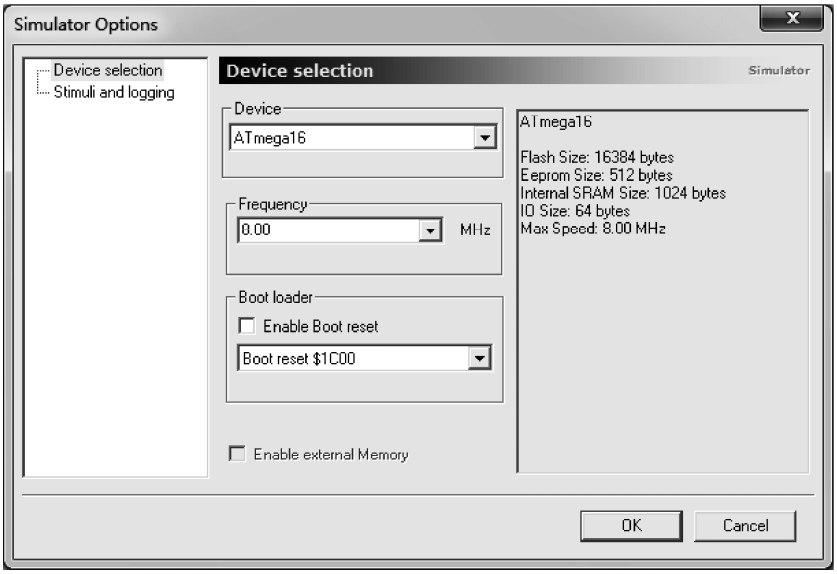


图 2-13 “Simulator Options”对话框（“Device selection”选项卡）

【“Device selection”选项卡】如图 2-13 所示，在“Device”栏中选择所需的单片机型号（在此选择“ATmega16”）；在“Frequency”栏中选择 AVR 单片机使用的晶振频率（在此选择 8MHz）；在“Boot loader”区域中可引导加载程序的设置，选中“Enable Boot reset”复选框时表示允许启动复位，在其下拉列表中可选择启动复位地址。

【Stimuli and logging 选项卡】如图 2-14 所示，在此可设置外部激励或记录端口的相关信息。在“Port”中选择外部激励端口或记录端口；在“Function”栏中选择外部激励端口（Stimuli）或选择记录端口（Logging）。

设置为外部激励（Stimuli）时，该端口必须为输入状态时有效，且还需指出激励文件位置，即在“File”栏中打开激励文件。激励文件中的值将在指定的周期被放在指定端口的 PIN 寄存器中，激励文件的格式与端口记录格式相同。

设置为记录端口（Logging）时，该端口必须为输出状态时有效，且还需在“File”栏中选择一个文件用于放置记录数据。文件上的内容是端口寄存器的内容。如果一个周期中端口寄存器的内容没有改变，就没有输出产生。记录文件在每次程序复位时被删除，在每次程序装入 AVR Studio 时，记录都将人工激活。

3．使用 AVR Studio 进行程序调试及软件仿真

在 AVR Studio 中进行模拟仿真时，其调试算法、程序流程等方面与硬件仿真机没有多大区别。对于 I/O 端口、定时器、UART、中断响应等方面，在 AVR Studio 中也可进行模拟

仿真。但是，由于硬件仿真机没有提供程序运行时间等方面的参数调试，所以 AVR Studio 在调试程序、计算一段程序运行所花的时间等方面比硬件仿真更加方便。

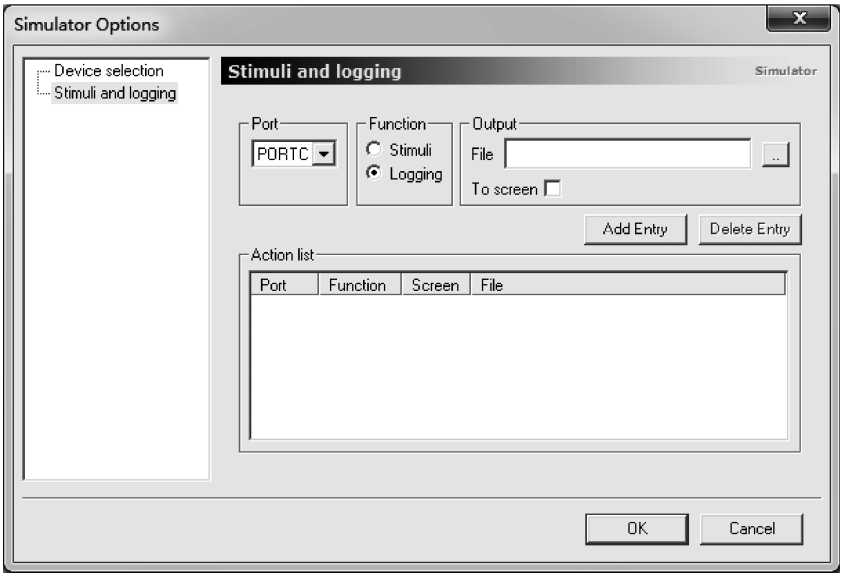


图 2-14 “Simulator Options”对话框（“Stimuli and logging”选项卡）

程序调试

在 AVR Studio 的“Debug”（调试）菜单栏中可选择开始调试、停止调试、运行、暂停、设置断点、单步、自动单步等操作。

【开始调试（Start Debugging）】若源程序在 CodeVisionAVR IDE 中，通过执行菜单命令“Tools”→“Debugger”导入到 AVR Studio，则在 AVR Studio 中的程序处于开始调试状态。

如果源程序是在 AVR Studio 中直接编写的，则在 AVR Studio 中先执行菜单命令“Build”将其编译，若编译程序没有错误，再执行菜单命令“Debug”→“Start Debugging”，或者在工具栏中单击图标，还可以按“Ctrl”+“Shift”+“Alt”+“F5”键，源程序进入开始调试状态。

在开始调试状态下，将链接调试平台，装载目标文件并执行复位操作，而所有的调试控制命令处于有效。通常，在调试模式下不能编辑程序。

【停止调试（Stop Debugging）】如果源程序进入了调试状态，在 AVR Studio 中执行菜单命令“Debug”→“Stop Debugging”，或者在工具栏中单击图标，还可以按“Ctrl”+“Shift”+“F5”键，将停止调试过程，并断开与调试平台的链接，进入源程序的编辑模式。

【运行（RUN）】如果源程序进入了调试状态，在 AVR Studio 中执行菜单命令“Debug”→“RUN”，或者在工具栏中单击图标，还可以按“F5”键，将启动（或重新启动）程序，使源程序一直运行到被用户停止或遇到一个断点。

【暂停（Break）】如果源程序进入调试运行状态，在 AVR Studio 中执行菜单命令“Debug”→“Break”，或者在工具栏中单击图标，还可以按“Ctrl”+“F5”键，将停止

程序运行。当源程序停止时，所有窗口中的信息都将更新。

【复位（Reset）】如果源程序进入了调试状态，在 AVR Studio 中执行菜单命令“Debug”→“Reset”，或者在工具栏中单击图标，还可以按“Shift”+“F5”键，将使源程序复位。当程序正在运行时，执行此命令则程序停止运行。

【设置断点（Toggle Breakpoint）】如果源程序进入了调试状态，在代码前可以设置断点。首先在需要设置断点的代码前单击鼠标左键，然后执行菜单命令“Debug”→“Toggle Breakpoint”，或者在工具栏中单击图标，还可以按“F9”键，即可将该行源代码设置为断点。

【单步执行（Step Into）】单步执行仅执行一条指令。如果源程序进入了调试状态，在 AVR Studio 中执行菜单命令“Debug”→“Step Into”，或者在工具栏中单击图标，还可以按“F11”键，源程序的一条指令会被执行。

【单步运行到断点（Step Over）】单步运行到断点仅执行一条指令，但是如果这条指令包含或调用了一个函数/子程序时，则该函数/子程序也会同时执行。如果在运行过程中遇到用户设置的断点，则程序运行将被挂起。如果源程序进入了调试状态时，在 AVR Studio 中执行菜单命令“Debug”→“Step Over”，或者在工具栏中单击图标，还可以按“F10”键，源程序的一条指令会被执行。

【跳出（Step Out）】跳出命令会使程序一直运行，直到当前函数结束。如果遇到用户设置的断点，程序运行将被挂起。当程序处在最外层（如主函数），执行菜单命令“Debug”→“Step Out”，或者在工具栏中单击图标，或按“Shift”+“F11”键时，则程序将继续运行，直到遇到一个断点或被用户停止。

【运行到光标处（Run To Cursor）】如果源程序进入了调试状态，在 AVR Studio 中执行菜单命令“Debug”→“Run To Cursor”，或者在工具栏中单击图标，或者按“Ctrl”+“F10”键时，将使程序运行到源代码编辑窗口中光标处停止。此时如果遇到用户的断点，程序的运行将不会被挂起。如果程序运行永远达不到光标指示处的语句，程序将一直继续运行，直到被用户停止。当此命令结束后，所有窗口中的信息将更新。由于这种方式依赖光标放置的位置，所以只能在 C 语言窗口方式时才能使用。

【自动运行（Auto Step）】自动运行能够重复执行指令。如果源程序进入了调试状态时，在 AVR Studio 中执行菜单命令“Debug”→“Auto Step”，或者在工具栏中单击图标，或者按“Ctrl”+“F10”键时，程序的运行将一直持续单步运行，直到遇到一个用户设置的断点或被用户停止。



软件仿真

源程序在调试运行过程中，通过相应的窗口可以观察各个寄存器的值，从而分析程序是否达到设计要求。

若需观察已导入到 AVR Studio 的“新建项目”源程序的运行状况时，首先在 I/O 观察窗口中单击“PORTC”（因为源程序使用了 PORTC 端口），然后执行自动运行命令时，可以看到“Processor”窗口中的 Cycle Counter 和 Stop Watch 中的数据不断更新，而 I/O 观察窗口中 PORTC 端口的状态值（Value）每隔 1s 将发生一次变化，其运行仿真如图 2-15 所示。若想终止运行，执行菜单命令“Debug”→“Reset”即可。

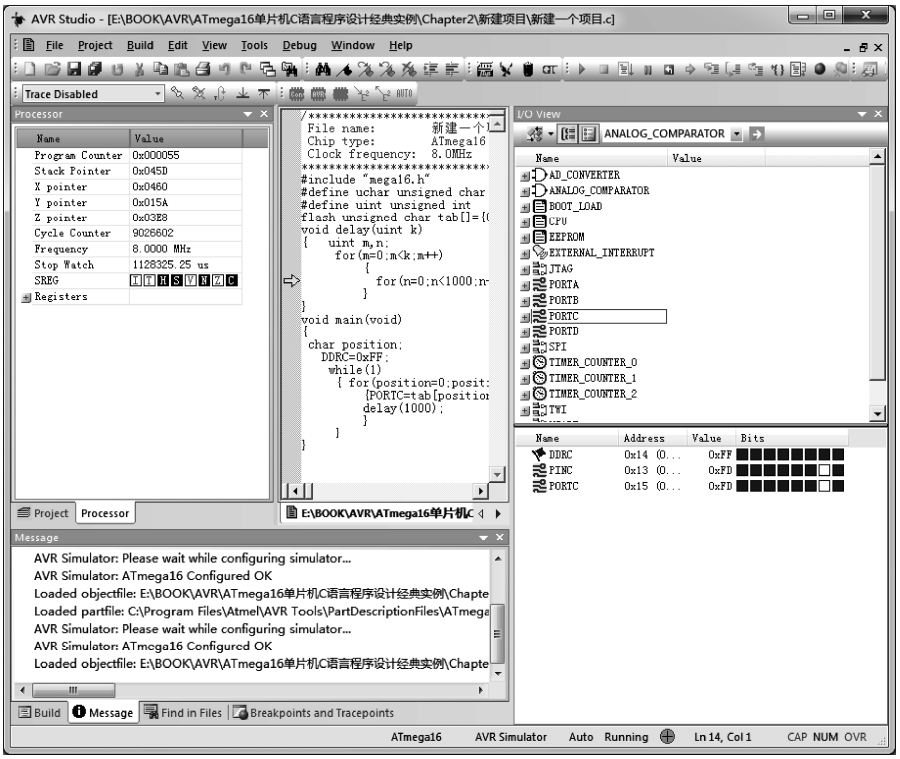


图 2-15 “新建项目”的运行仿真图

任务 3　PonyProg2000 下载软件的使用

PonyProg2000 是一款功能强大的 ISP 下载软件，它可以下载/编程 AVR、PIC 等系列的单片机和很多类型的 E^2PROM。由于该软件在启动时会出现一只马头图案，因此有人将该软件又称为小马软件。

使用 PonyProg2000 软件将已编译生成的 Hex 文件下载到 AVR 单片机中时，一般需要按下述步骤进行操作。

【步骤 1：启动 PonyProg2000 软件】执行菜单命令“开始”→“程序”→“PonyProg”→“PonyProg2000”，将进入图 2-16 所示的 PonyProg2000 的启动界面。在 About PonyProg2000 界面中选择“Disable sound”复选框，将取消音效“马叫声”。

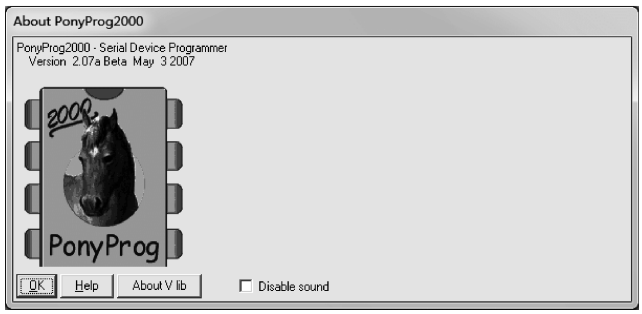


图 2-16　PonyProg2000 启动界面

【步骤 2：选择下载器件型号】在 About PonyProg2000 界面上单击“OK”按钮后，在 PonyProg2000 界面执行菜单命令“Device” → “AVR Auto” → “ATmega16”，选择下载目标芯片型号为“ATmega16”。

【步骤 3：端口设置】在 PonyProg2000 界面执行菜单命令“Setup” → “Interface Setup”，将弹出图 2-17 所示的“I/O port setup”对话框。在此对话框中可进行下载方式设置及选择控制线的极性，然后单击“OK”按钮。PonyProg2000 支持的下载方式有并行下载和串行下载。

【步骤 4：端口校正】端口设置好后，在 PonyProg2000 界面执行菜单命令“Setup” → “Calibration”，将弹出图 2-18 所示的对话框。在此对话框中单击“Yes”按钮开始端口校正。校正完成后，会弹出一提示窗口，单击“OK”按钮即可。

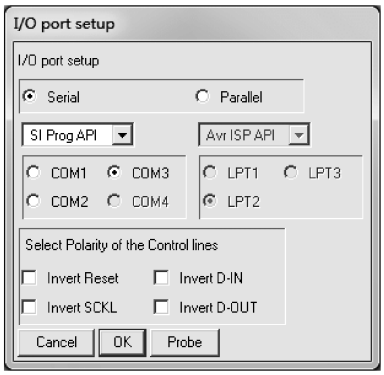


图 2-17 “I/O port setup”对话框

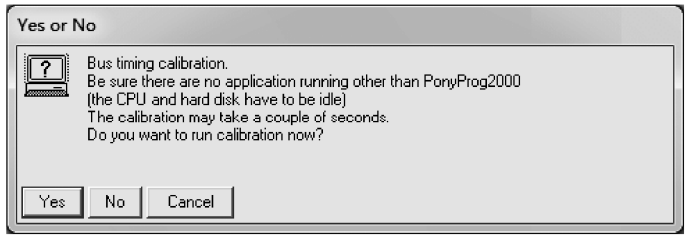


图 2-18 端口校正

【步骤 5：配置熔丝位】在 PonyProg2000 界面执行菜单命令“Command” → “Security and Configuration Bits”，可进行熔丝位的配置，如图 2-19 所示。选中熔丝位时，表示其为“0”状态，其配置方法请参照 CVAVR 中对目标设备芯片编程的相关内容。

【步骤 6：编程设置】AVR 单片机的程序存储器是 Flash 存储器，在下载程序前必须先对其进行擦除操作。在 PonyProg2000 界面执行菜单命令“Command” → “Program Options”，在弹出的“Program Options”对话框中选中“Reload Files”、“Erase”、“Write Program memory（FLASH）”3 个复选框，如图 2-20 所示，然后单击“OK”按钮。

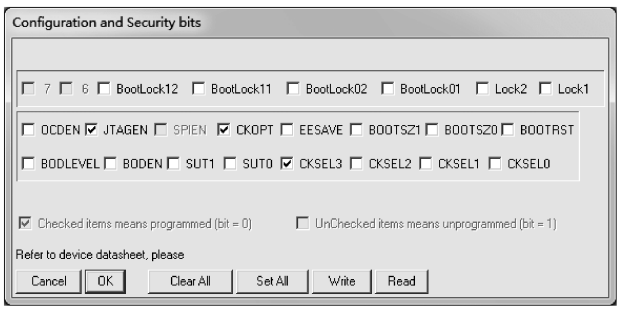


图 2-19 “Configuration and Security bits”对话框

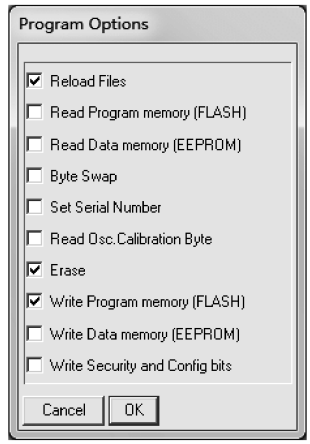


图 2-20 “Program Options”对话框

【步骤 7：装载编程文件】在 PonyProg2000 界面执行菜单命令“File”→“Open Program(FLASH)Files”，弹出“Open Program(FLASH) content file”对话框。在此对话框中指定文件类型为“*.hex”，在查找范围中找到已通过 CodeVisionAVR IDE 编译并生成的“新建项目.hex”文件后，PonyProg2000 的编辑窗口中可看到已装载了编程文件，如图 2-21 所示。

【步骤 8：下载程序】在 PonyProg2000 界面执行菜单命令“Command”→“Program”，可将图 2-21 中装载的编程文件下载 ATmega16 中。如果删除、下载、确认等操作没有异常，就会弹出“Program successful”消息窗口，表示下载程序成功。

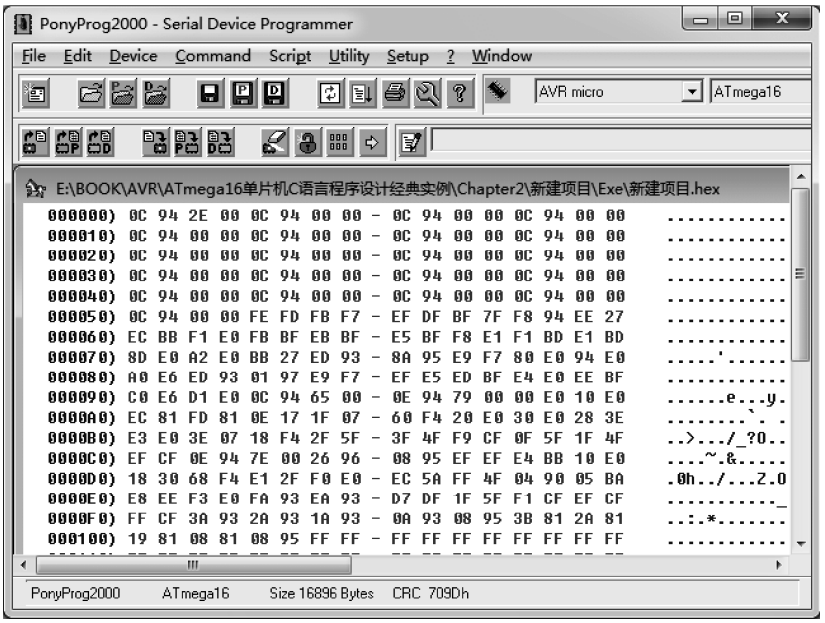


图 2-21　装载编程文件

任务 4　Proteus 7.10 仿真软件的使用

在 ATmega16 单片机的学习与开发过程中，CVAVR 是程序设计开发平台，它能进行程序的编译；AVR Studio 可进行程序的编译、调试与软件仿真，但是它们都不能直接进行硬件仿真。Proteus 软件具有交互式仿真功能，它不仅是模拟电路、数字电路、模/数混合电路的设计与仿真平台，更是目前世界上最先进、最完整的多种型号微处理器系统的设计与仿真平台。如果将 CVAVR 等 AVR 编译软件和 Proteus 软件有机结合起来，那么 ATmega16 单片机的设计与开发将在软、硬件仿真上得到完美的结合。

1．Proteus 软件的使用

Proteus 软件由 ISIS（Intelligent Schematic Input System）和 ARES（Advanced Routing and Editing Software）两个软件构成，其中 ISIS 是一款智能原理图输入系统软件，可作为电子系统仿真平台；ARES 是一款高级布线编辑软件，用于制作印制电路板（PCB）。由于篇幅的原因，Proteus ISIS 的详细介绍及 Proteus ARES 的使用方法，本书不进行介绍，感兴趣

的读者可以参考本书作者编写的《基于 Proteus 的 AVR 单片机 C 语言程序设计与仿真》一书。图 2-22 所示的是与本章任务 1 中的“新建一个项目.c”源程序对应原理图，现以此图为例，简单介绍 Proteus ISIS 的使用方法。

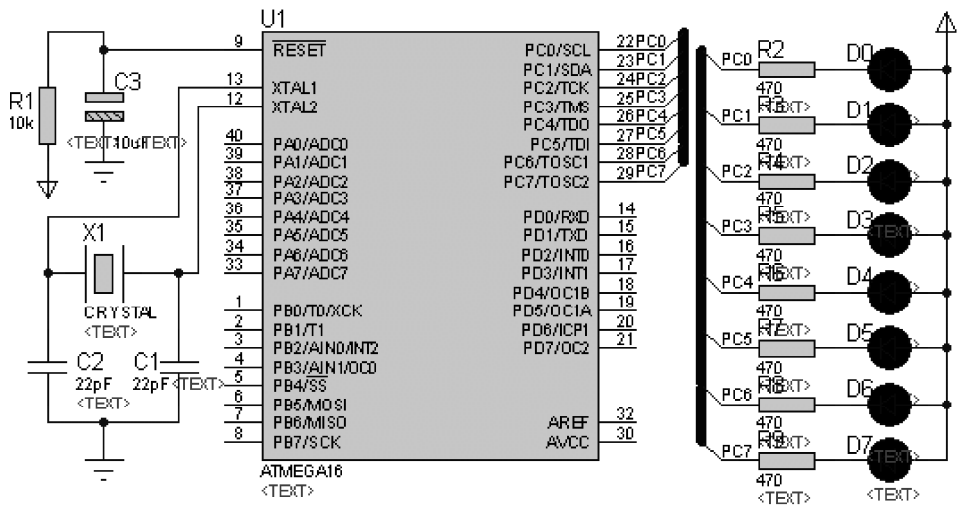


图 2-22　与“新建一个项目.c”对应的原理图

1）新建设计文件　在桌面上双击图标，打开 ISIS 7 Professional 窗口。执行菜单命令“File”→“New Design”，弹出如图 2-23 所示的模板选择窗口。横向图纸为“Landscape”，纵向图纸为“Portrait”，“DEFAULT”为默认模板。选中“DEFAULT”，单击“OK”按钮，则新建了一个 DEFAULT 模板。也可以在 ISIS 7 Professional 窗口中直接单击图标，也可新建一个 DEFAULT 模板。

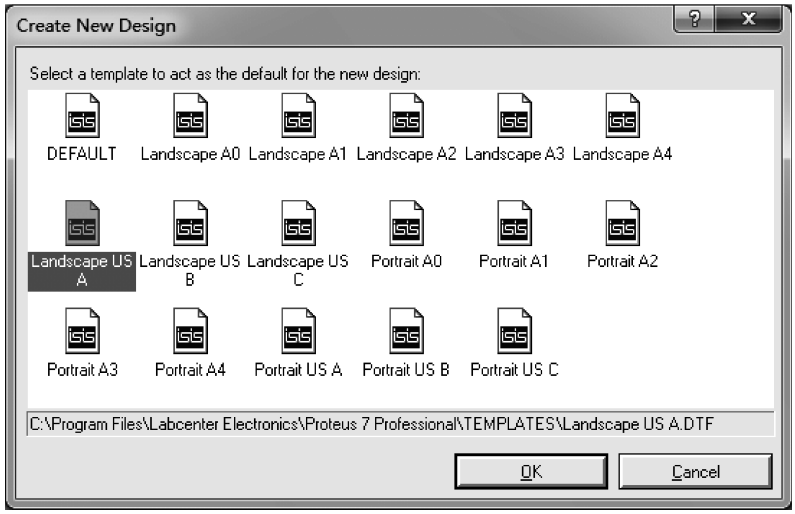


图 2-23　图纸模板选择

2）设定图纸的大小　执行菜单命令“System”→“Set Junction Dots…”，在弹出的对话框中选择“A4”复选框，单击“OK”按钮，完成图纸的设置。图纸设置好后，进入如图 2-24 所示的 ISIS 7 Professional 窗口。

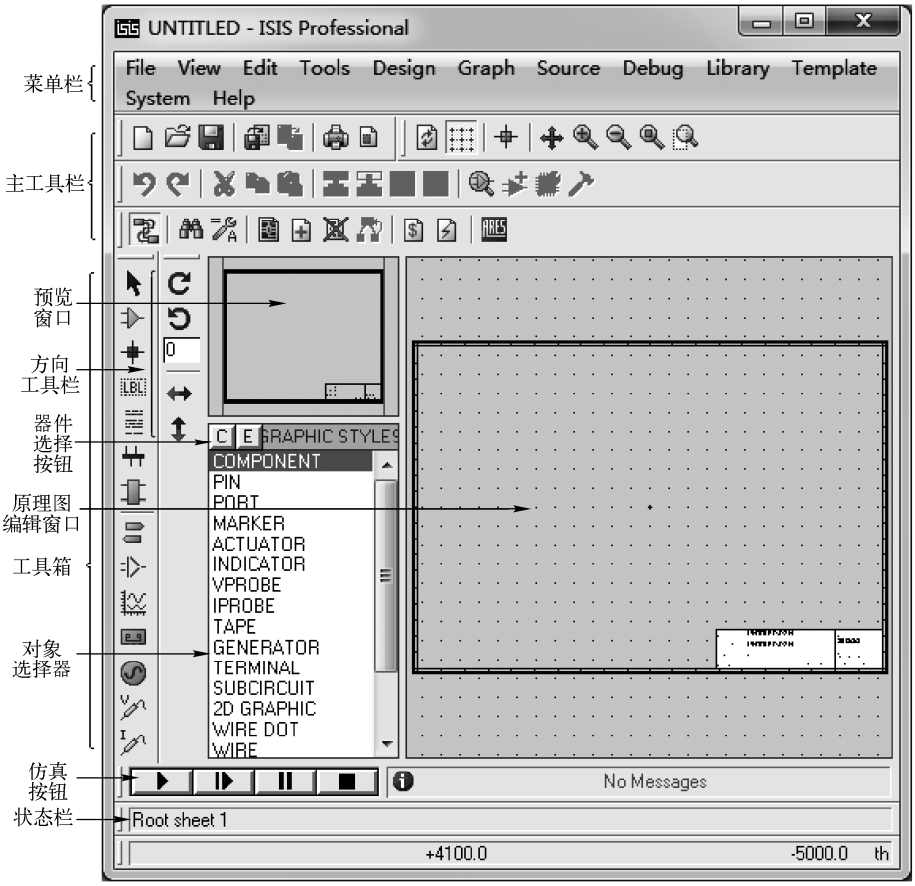


图 2-24　ISIS 7 Professional 窗口

3）保存设计项目　新建一个 DEFAULT 模板后，在 ISIS 7 Professional 窗口的标题栏上显示为 DEFAULT。单击图标，或者执行菜单命令“File”→“Save Design…”，弹出“Save ISIS Design File”对话框。在此对话框中选择合适的保存路径，输入保存文件名为“新建一个项目”。该文件的扩展名为.DSN，即该文件名为“新建一个项目.DSN”。文件保存后，在 ISIS 7 Professional 窗口的标题栏上显示为“新建一个项目”。

4）为设计项目添加电路元器件　本例中使用的元器件见表 2-2。在元器件选择按钮 P L DEVICES 中单击“P”按钮，或执行菜单“Library”→“Pick Device/Symbol”，弹出图 2-25 所示的“Pick Devices”对话框。在此对话框中，添加元器件的方法有两种。

表 2-2　本例中使用的元器件

单片机 ATmega16	瓷片电容 CAP 22pF	晶振 CRYSTAL 8MHz	发光二极管 LED-GREEN
电解电容 CAP-ELEC	发光二极管 LED-RED	发光二极管 LED-BLUE	发光二极管 LED-YELLOW
电阻 RES			

☺ 在“keywords”栏中输入元器件名称，如“ATmega16”，则出现与关键字匹配的元器件列表，如图 2-26 所示。选中并双击“ATmega16”所在行后，单击“OK”按钮或按“Enter”键，便将器件 ATmega16 加入到 ISIS 对象选择器中。

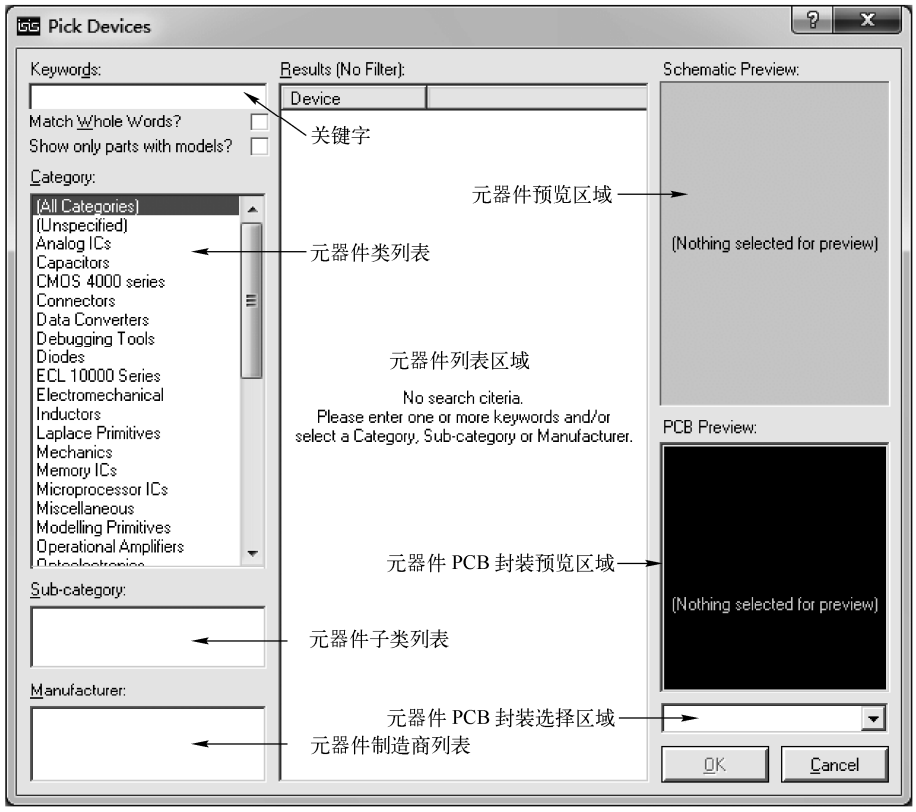


图 2-25 “Pick Devices”对话框

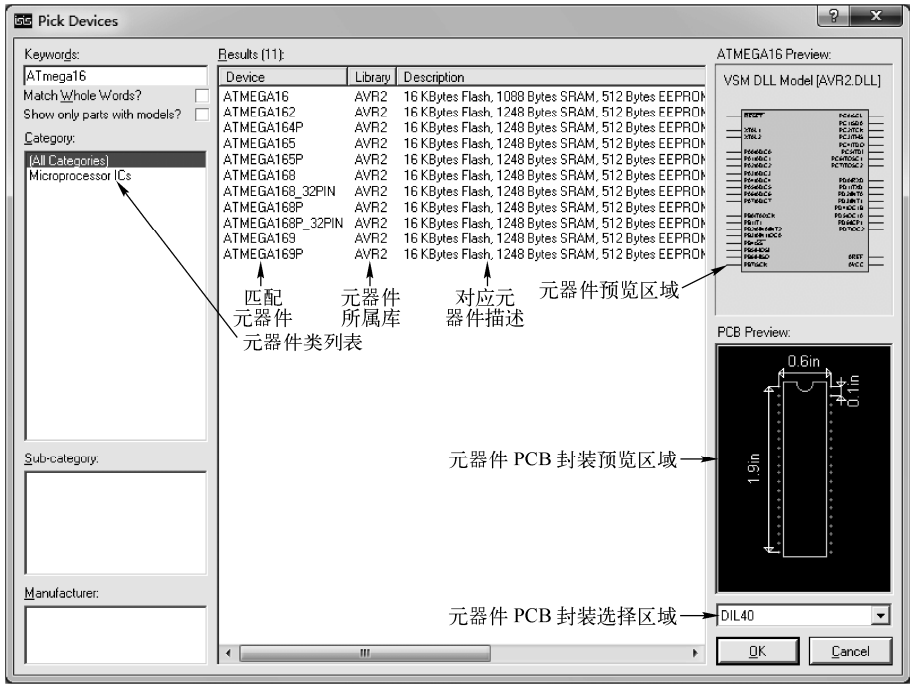


图 2-26 输入元器件名称后的“Pick Devices”对话框

☺ 在元器件类列表中选择元器件所属类，然后在子类列表中选择所属子类，同时当对元器件的制造商有要求时，在制造商区域选择期望的厂商，即可在元器件列表区域得到相应的元器件。

按照以上方法将元器件添加到 ISIS 对象选择器中。

5）放置、移动、旋转、删除对象　元器件添加到 ISIS 对象选择器中后，在对象选择器中单击要放置的元器件，蓝色条出现在该元器件名上，再在原理图编辑窗口中单击，即可放置一个元器件。也可以在按住鼠标左键的同时移动鼠标，在合适位置释放，将元器件放置在预定位置。

在原理图编辑窗口中若要移动元器件或连线时，先用鼠标右键单击对象，使元器件或连线处于选中状态（默认情况下为红色），再按住鼠标左键拖动，元器件或连线就跟随指针移动，到达合适位置时，松开鼠标即可。

放置元器件前，单击要放置的元器件，蓝色条出现在该元器件名上，单击方向工具栏上相应的转向按钮可旋转元器件，再在原理图编辑窗口中单击即可放置一个已经更改方向的元器件。若在原理图编辑窗口中需要更改元器件方向时，单击选中该元器件，再单击块旋转图标，在弹出的对话框中输入旋转的角度即可实现更改元器件方向。

在原理图编辑窗口中要删除元器件时，用鼠标右键双击该元器件即可删除该元器件，或者先用鼠标左键单击选中该元器件，再按“Delete”键也可删除元器件。

通过放置、移动、旋转、删除元器件后，可将各元器件放置在 ISIS 原理图编辑窗口的合适位置，如图 2-27 所示。

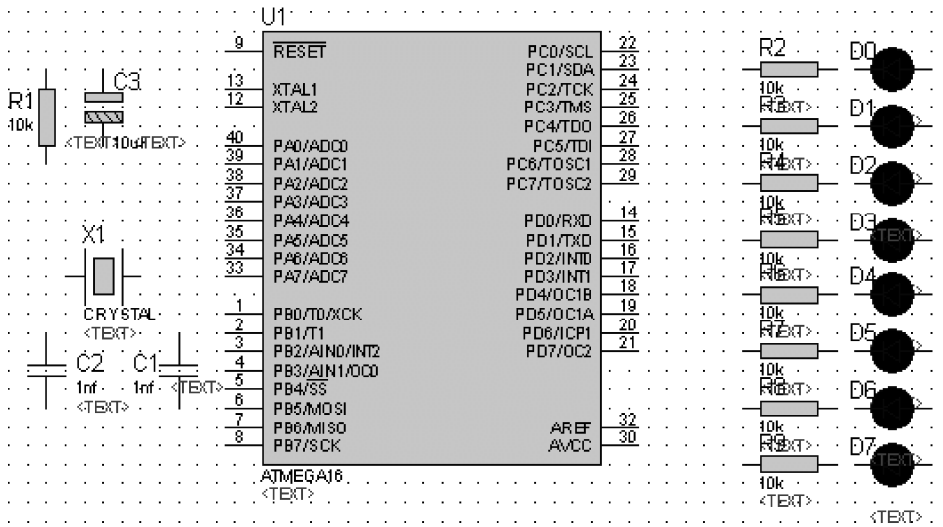


图 2-27　各元器件放置在原理图编辑窗口的合适位置

6）放置电源、地　单击工具箱中元器件终端图标，在对象选择器中单击“POWER”，使其出现蓝色条，再在原理图编辑窗口合适位置单击鼠标就将电源放置在原理图中。同样，在对象选择器中单击 GROUND，再在原理图编辑窗口合适位置单击鼠标就将“地”放置在原理图中。

7）布线　在 ISIS 原理图编辑窗口中没有专门的布线按钮，但系统默认自动布线有效，因此可直接绘制连线。

【在两个对象间连线】

（1）将光标靠近一个对象引脚端，该处自动出现一个“”，单击鼠标左键。

（2）拖动鼠标，将光标移至另一对象的引脚端，在该引脚端出现一个“”时再单击鼠标即可绘制一根连线，如图 2-28（a）所示；若想手动设定布线路径时，拖动鼠标，在想要的拐点处单击，设定布线路径，到达绘制线端的另一端后单击鼠标左键，就绘制好一根连线，如图 2-28（b）所示。在拖动鼠标过程中，按住“Ctrl”键，在绘制线的另一端出现一个“”时单击鼠标左键，可手动绘制任意角度的连线，如图 2-28（c）所示。

【移动布线、更改线形】

（1）单击鼠标左键选中连线，将光标靠近该布线，在该布线出现双箭头，如图 2-28（d）所示。

（2）按住左键拖动鼠标，该布线就跟随移动。

（3）若多根线要同时移动时，先框选这些布线，再单击块移动按钮，拖动鼠标，在合适位置单击鼠标左键，即可改变布线的位置。

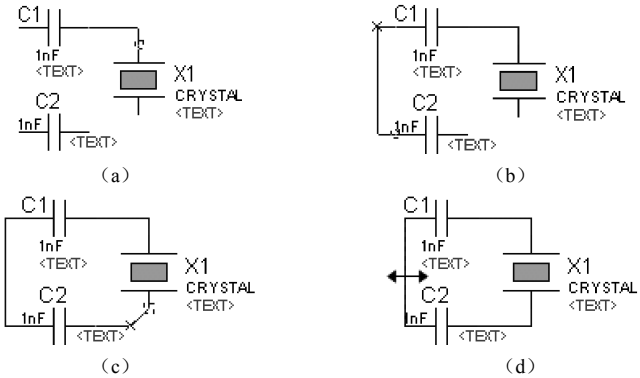


图 2-28　布线

【总线及分支线的绘制方法】

（1）将光标靠近一个对象引脚端，该处自动出现一个“”，单击鼠标左键。

（2）拖动鼠标，在合适位置双击鼠标左键，绘制出一条直线。

（3）如果该线为单线，要设置为总线时，先选中该线，用鼠标右键单击，在弹出的菜单中选择“Edit Wire Style”，如图 2-29（a）所示，再在弹出的对话框中的“Global Style”栏中选择“BUS WIRE”，如图 2-29（b）所示。绘制的总线如图 2-30 所示。

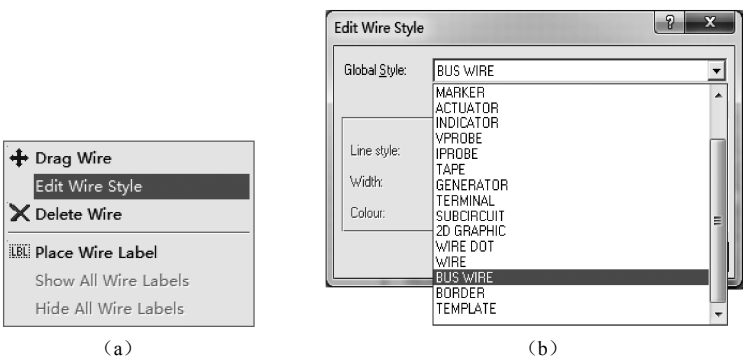


图 2-29　总线绘制方法

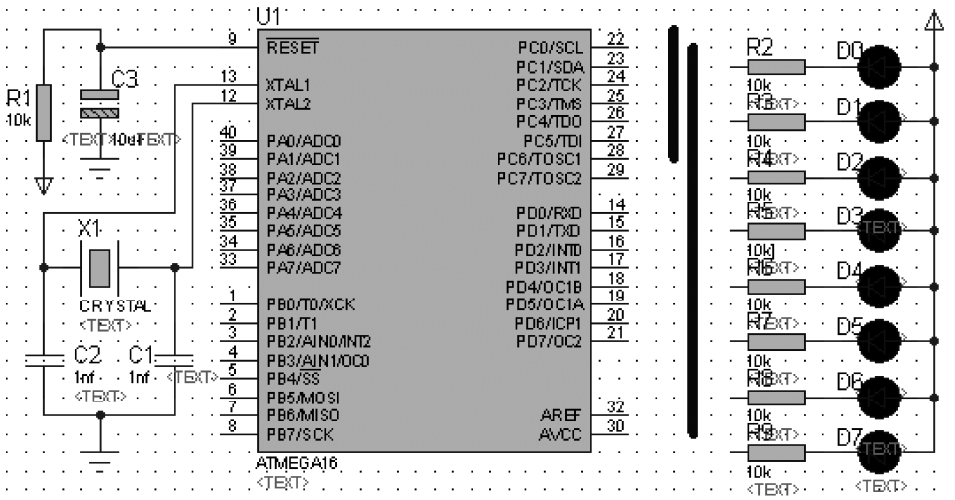


图 2-30　绘制的总线

总线分支线绘制步骤如下所述。

（1）将光标靠近一个对象引脚端，该处自动出现一个“”，单击鼠标左键。

（2）拖动鼠标，在靠近总线合适位置双击鼠标左键，绘制出一条直线。

（3）将光标靠近该直线末端，该处自动出现一个“”，单击鼠标左键。

（4）拖动鼠标，同时按住“Ctrl”键，在总线出现一个“”时单击鼠标左键，即可绘制好一根分支线，如图 2-31 所示。

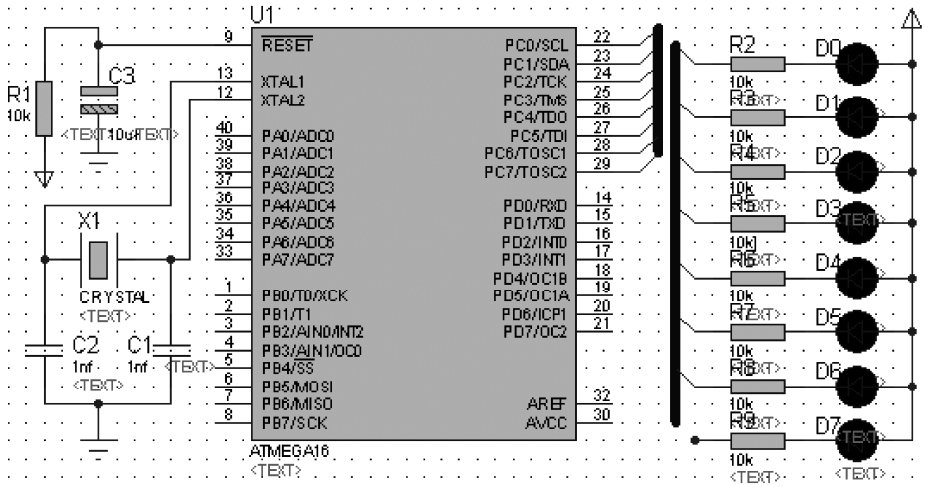


图 2-31　分支线的绘制方法

（5）在工具箱中单击图标，再在总线或各分支线上单击鼠标左键，弹出如图 2-32 所示的“Edit Wire Label”对话框。在“Label”选项卡的“String”栏中输入相应的线路标号，如总线为“AD[0..7]”（表示有 AD0～AD7 共 8 根数据线），分支线为 AD0、AD1 等。

8）设置、修改元器件属性　在需要修改属性的元器件上单击鼠标右键，在弹出的菜单中选择“Edit Properties”，或者按“Ctrl”+“E”键，将出现“Edit Component”对话框，在此对话框中设置相关信息。例如，修改电容为 22pF，如图 2-33 所示。

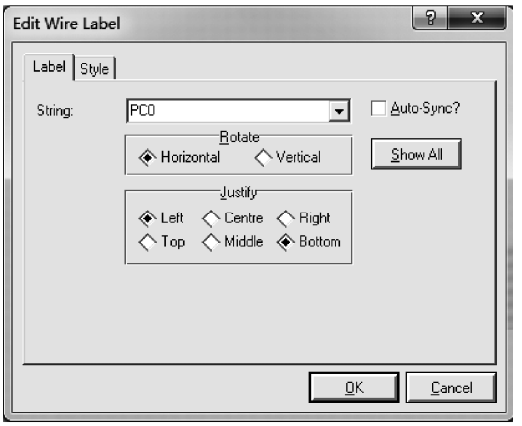


图 2-32 “Edit Wire Label”对话框

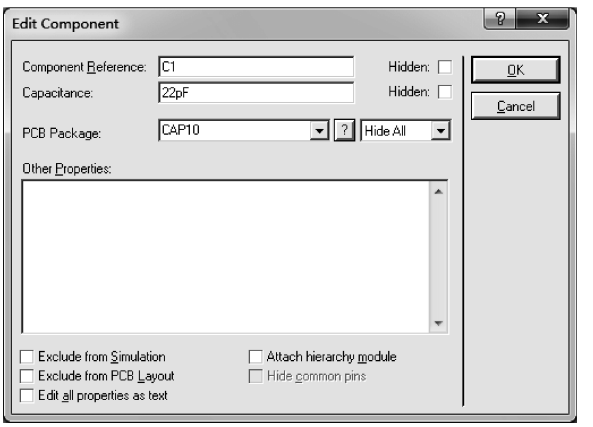


图 2-33 “Edit Component”对话框

9）编辑设计原理图界面 根据以上步骤及方法在原理图编辑窗口中绘制完图 2-22 所示的电路图后，可以将不需要显示的一些项目隐藏起来，把界面编辑成简洁、清爽的界面。执行菜单命令“View”→“Grid”，可以去掉界面中的网格；执行菜单命令“Template”→“Set Design Defaults”，在弹出的“Edit Design Defaults”对话框中取消“Show hidden text?”选项的选中状态，可以去掉元器件的文本内容。编辑好的单片机仿真原理图如图 2-34 所示。

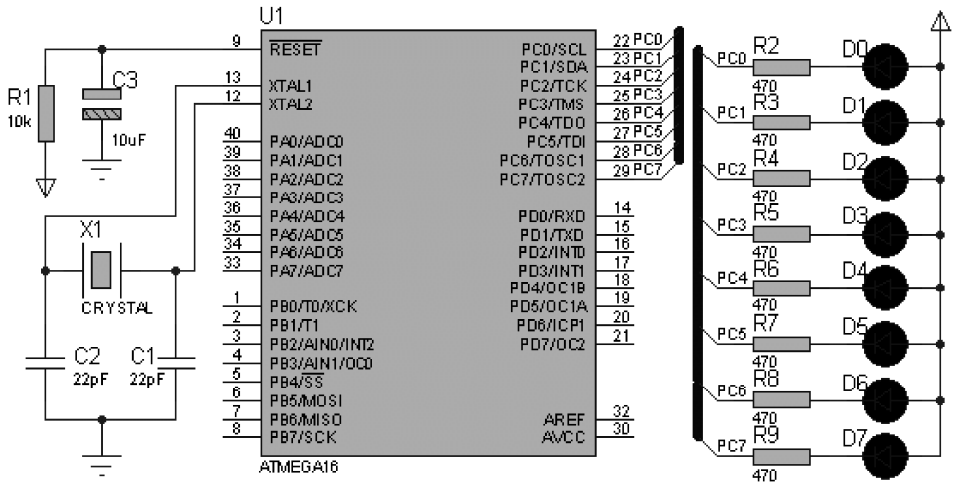


图 2-34 编辑好的单片机仿真原理图

10）单片机程序仿真 在原理图中双击 ATmega16 单片机，将弹出“Edit Component”对话框。在“Program File”栏中单击按钮，添加任务 1 中由 CVAVR 生成的“新建项目.coff”或“新建项目.hex”文件。在“Clock Frequency”选项中设置单片机的工作频率为 8MHz，如图 2-35 所示。设置好后，单击“OK”按钮将原理图保存，并回到原理图编辑界面。

在原理图编辑界面中单击仿真按钮，即可进行单片机程序仿真。

> 注意：
>
> 在仿真过程中，元器件的某些引脚显示红色的小方点表示该引脚为高电平状态，引脚显示蓝色的小方点表示该引脚处于低电平状态。仿真效果如图 2-36 所示。

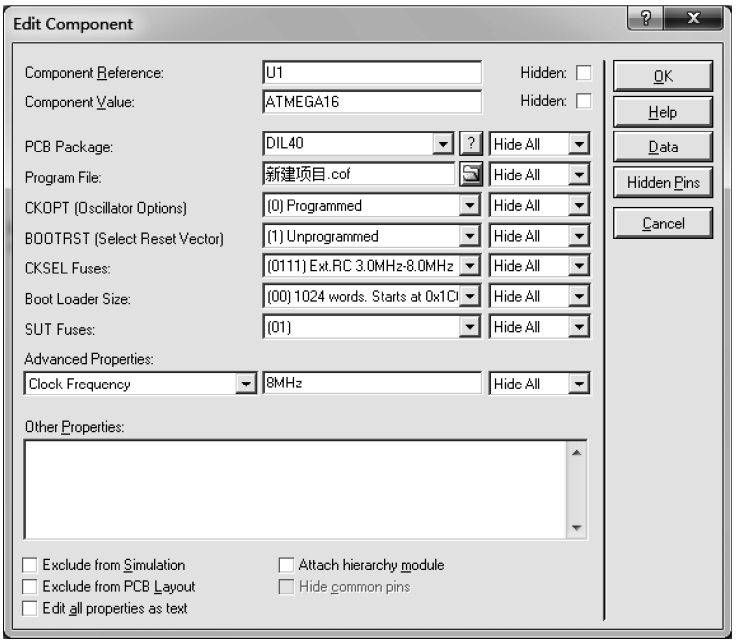


图 2-35　单片机编辑对话框设置

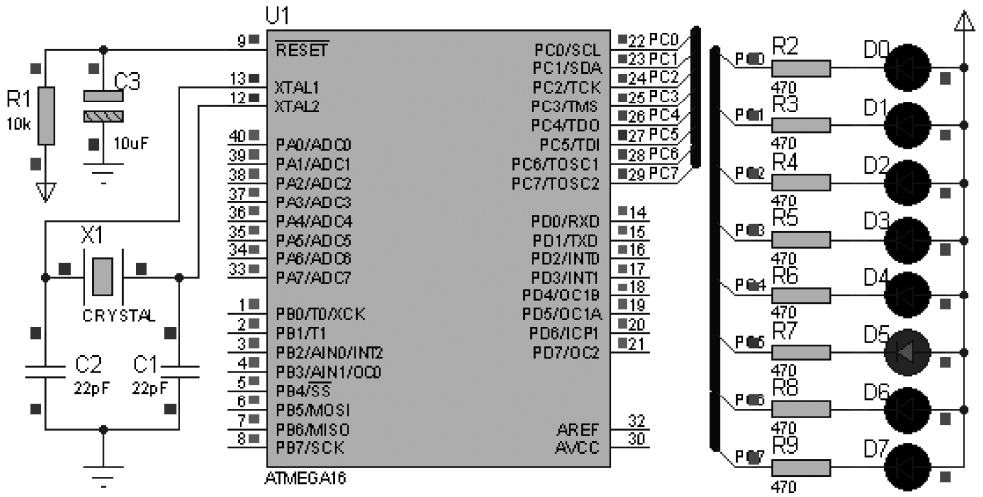


图 2-36　“新建项目”的仿真效果图

项目三　AVR 单片机 C 语言编程基础

〖知识目标〗
☺ 了解 CVAVR 的 C 语言程序结构;
☺ 掌握 CVAVR 的 C 语言标志符和关键字、数据类型、常量、变量、运算符及表达式;
☺ 掌握 CVAVR 的 C 语言选择、循环流程控制语句;
☺ 掌握 CVAVR 的 C 语言的数组、指针、函数、预编译的使用;
☺ 能够使用 CVAVR 的 C 语言进行应用程序设计。
〖能力目标〗
☺ 掌握 CVAVR 的 C 语言基本语法、基本概念、程序设计的一般方法;
☺ 能运用所学知识和技能对一般问题进行分析，并能编制出高效的 C51 应用程序。

对于单片机应用技术而言，除了要学习单片机系统的硬件设计外，还要学习单片机的编程语言。对于 ATmega16 单片机来说，其编程语言主要是汇编语言和 C 语言这两种。由于汇编语言具有可移植性和可读性差，以及本身就是一种编程效率低下的低级语言等特点，因此利用它进行编程和维护极不方便。而使用 C 语言进行嵌入式系统的开发，有着汇编语言编程不可比拟的优势，因此在单片机应用系统中，通常将 C 语言作为系统开发的首选语言。

有很多第三方厂商为 AVR 单片机开发了用于程序开发的 C 编译器，这些 C 编译器的语法结构基本相同，但在某些细节上还是有所区别，因此当用户选择了某个 C 编译器后，应掌握相应的 C 语言语法。本书将以 CVAVR C 编译器（简称 CVAVR）为例，讲述 AVR 单片机的开发及应用。



任务 1　C 语言程序组成的识读

C 语言是国际上广泛流行的计算机高级语言，它是一种源于编写 UNIX 操作系统的结构化语言，程序有清晰的层次结构。

1．C 语言程序结构

AVR 单片机的 C 语言程序结构与一般 C 语言有很大的区别，每个 AVR 单片机的 C 语言程序至少有一个 main()函数（即主函数）且只能有一个，它是 C 语言程序的基础，是程序代码执行的起点，而其他函数都是通过 main()函数直接或间接调用的。

〖AVR 单片机的 C 语言程序结构的特点〗
☺ 一个 C 语言源程序由一个或多个源文件组成，主要包括一些 C 源文件（即后缀名

为“.C”的文件）和头文件（即后缀名为“.h”的文件）。对于一些支持 C 语言与汇编语言混合编程的编译器而言，它还可包括一些汇编源程序（即后缀名为“.asm”的文件）。

☺ 每个源文件至少包含一个 main()函数，也可以包含一个 main()函数和其他多个函数。头文件中声明一些函数、变量或预定义一些特定值，而函数是在 C 源文件中实现的。

☺ 一个 C 语言程序总是从 main()函数开始执行的，而不论 main()函数在整个程序中的位置如何。

☺ 源程序中可以有预处理命令（如 include 命令），这些命令通常放在源文件或源程序的最前面。

☺ 每个声明或语句都以分号结尾，但预处理命令、函数头和花括号“{}”后不能加分号。

☺ 标志符、关键字之间必须加一个空格以示间隔。若已有明显的间隔符，也可不再加空格来间隔。

☺ 源程序中所用到的变量都必须先声明，然后才能使用，否则编译时会报错。

C 源程序的书写格式自由度较高，灵活性很强，有较大的任意性，但这并不表示 C 源程序可以随意乱写。为了书写清晰，并便于阅读、理解、维护，在书写程序时最好遵循以下规则。

☺ 通常情况下，一个声明或一个语句占用一行。在语句的后面可适量添加一些注释，以增强程序的可读性。

☺ 不同结构层次的语句从不同的起始位置开始，即在同一结构层次中的语句，应缩进同样的字数。

☺ 用“{}”括起来的部分表示程序的某一层次结构。“{}”通常写在层次结构语句第 1 个字母的下方，与结构化语句对齐，并占用一行。

在此以下面的程序为例，进一步说明 AVR 单片机 C 语言的程序结构特点及书写规则。程序清单如下所示。

```
/*****************************************************          //第 1 行
 File name:          例 1.C                                     //第 2 行
 Chip type:          ATmega16                                   //第 3 行
 Clock frequency:    8.0MHz                                     //第 4 行
*****************************************************/          //第 5 行
#include <mega16.h>                                             //第 6 行
#define uint unsigned int                                       //第 7 行
void delay(uint k)                                              //第 8 行
{                                                               //第 9 行
      uint   m,n;                                               //第 10 行
      for(m=0;m<k;m++)                                          //第 11 行
      {                                                         //第 12 行
          for(n=0;n<1000;n++);                                  //第 13 行
      }                                                         //第 14 行
}                                                               //第 15 行
void main(void)                                                 //第 16 行
{                                                               //第 17 行
  DDRC=0xFF;                                                    //第 18 行
    while(1)                                                    //第 19 行
```

```
    {                                                    //第 20 行
      PORTC.0=～PORTC.0;                                 //第 21 行
       delay(1000);                                      //第 22 行
    }                                                    //第 23 行
}                                                        //第 24 行
```

这个小程序的作用是让接在 ATmega16 的 PC0 端口上的 LED 进行秒闪显示。下面分析这个 C 程序源代码。

第 1 行至第 5 行为注释部分。传统的注释定界符使用斜杠-星号（即“/*”）和星号-斜杠（即“*/”）。斜杠-星号用于注释的开始。编译器一旦遇到斜杠-星号（即“/*”），就忽略后面的文本（即使是多行文本），直到遇到星号-斜杠（即“*/”）为止。简而言之，在此程序中，第 1 行至第 5 行的内容不参与编译。在程序中还可使用双斜杠（即“//”）来作为注释定界符。若使用双斜杠（即“//”），则编译器会忽略该行语句中双斜杠（即“//”）后面的一些文本。

第 6 行和第 7 行分别是两条不同的预处理命令。在程序中，凡是以“#”开头的均表示这是一条预处理命令语句。第 6 行为文件包含预处理命令，其意义是把双引号（即“”）或尖括号（<>）内指定的文件包含到本程序中，成为本程序的一部分。第 7 行为宏定义预处理命令语句，表示 uint 为无符号整数类型。被包含的文件通常是由系统提供的，也可以由程序员自己编写，其后缀名为“.h”。C 语言的头文件中包括了各个标准库函数的函数原型。因此，在程序中调用一个库函数时，都必须包含函数原型所在的头文件。在 AVR 单片机中，不同的 C 语言编译器，其头文件也不尽相同。此例中，由于使用的是 CVAVR 编译器，所以头文件是“mega16.h”，这个文件里定义了 ATmega16 的各个寄存器。若使用 ICCAVR 编译器，则其头文件应为“iom16v.h”。

第 8 行定义了一个延时函数，其函数名为“delay”，函数的参数为“uint k”。该函数采用了两个层次结构和两重循环语句。第 9 行至第 15 行表示外部层次结构，其中第 9 行表示延时函数从此处开始执行，第 15 行表示延时函数的结束。第 12 行至第 14 行表示内部层次；第 10 行至第 14 行为数据说明和执行语句部分。第 11 行为外循环部分；第 13 行为内循环部分。

第 16 行定义了 main 主函数，函数的参数为“void”，意思是函数的参数为空，即不用传递给函数参数，函数即可运行。同样，该函数也采用了两个层次结构，第 17 行至第 24 行为外部层次结构；第 20 行至第 23 行为内部层次结构。

2．标志符与关键字

C 语言的标志符用于标志源程序中变量、函数、标号和各种用户定义的对象的名字。CVAVR C 中的标志符只能由字母（A～Z、a～z）、数字（0～9）、下画线（_）组成。其中第 1 个字符必须是字母或下画线，随后只能取字母、数字或下画线。标志符区分大小写，其长度不能超过 32 个字符。

> 注意:
> 标志符中不能使用中文。

关键字是由 C 语言规定的具有特定意义的特殊标志符，有时又称为保留字。关键字应当以小写形式输入。在编写 C 语言源程序时，用户定义的标志符不能与关键字相同。表 3-1 列出了 CVAVR C 中的一些关键字。

表 3-1　CVAVR C 中的一些关键字

关　键　字	用　　途	说　　明
auto	存储种类声明	用于声明局部变量
break	程序语句	退出最内层循环体
bit	位变量语句	位变量的值是 1(true)或 0(false)
case	程序语句	switch 语句中的选择项
char	数据类型声明	单字节整型或字符型数据
const	存储类型声明	在程序执行过程中不可修改的变量值
continue	程序语句	退出本次循环，转向下一次循环
default	程序语句	switch 语句中的失败选择项
defined	预处理语句	defined 预处理语句
do	程序语句	构成 do…while 循环结构
double	数据类型声明	双精度浮点数
eeprom	eeprom 存储器	eeprom 存储器地址空间
else	程序语句	构成 if…else 选择结构
enum	数据类型声明	枚举
extern	存储类型声明	在其他程序模块中声明的全局变量
flash	Flash 存储器	Flash 存储器空间
float	数据类型声明	单精度浮点数
for	程序语句	构成 for 循环结构
goto	程序语句	构成 goto 循环结构
if	程序语句	构成 if…else 选择结构
int	数据类型声明	基本整型数
interrupt	中断声明	声明 AVR 中断系统中使用的中断
long	数据类型声明	长整型数
register	存储类型声明	使用 CPU 内部的寄存器变量
return	程序语句	函数返回
short	数据类型声明	短整型数
signed	数据类型声明	有符号数，二进制的最高位为符号位
sizeof	运算符	计算表达式或数据类型的字节数
sfrb	寄存器声明	声明 sfrb 寄存器的地址
sfiw	寄存器声明	声明 sfrw 寄存器的地址
static	存储类型声明	静态变量
stuct	数据类型声明	结构类型数据
switch	程序语句	构成 switch 选择语句
typedef	数据类型声明	重新进行数据类型定义
union	数据类型声明	联合类型数据
unsigned	数据类型声明	无符号数据
void	数据类型声明	无返回数据
volatile	数据类型声明	声明该变量在程序执行中可被隐含改变
while	程序语句	构成 while 和 do…while 循环语句

3．数据类型

具有一定格式的数字或数值称为数据，数据是计算机操作的对象。数据的不同格式称为数据类型。

CVAVR C 支持的数据类型有位变量型（bit）、字符型（char）、无符号字符型（unsigned char）、有符号字符型（signed char）、整型（int）、短整型（short int）、无符号整型（unsigned int）、有符号整型（signed int）、长整型（long int）、无符号长整型（unsigned long int）、有符号长整型（signed long int）、单精度浮点型（float）、双精度浮点型（double）等，如图 3-1 所示。

基本类型就是使用频率最高的数据类型，其值不可以再分解为其他类型。CVAVR C 基本数据类型的长度和范围见表 3-2。若在 CVAVR 编译器中执行菜单命令“Project”→“Configure”，并在“Configure Project”对话框的“Code Generation”选项卡中将“char is unsigned”复选框选中，或者在源程序中使用了“#pragma uchar+”指令时，则 char 的范围为 0～255。

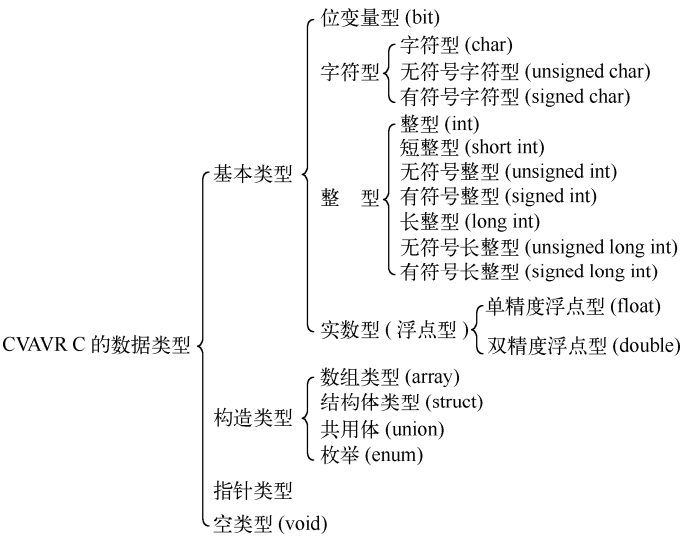


图 3-1　CVAVR C 支持的数据类型

表 3-2　CVAVR C 基本数据类型的长度和范围

类　型	长度/bit	长度/Byte	范　围
位变量型（bit）	1	…	0，1
字符型（char）	8	单字节	–128～127
无符号字符型（unsigned char）	8	单字节	0～255
有符号字符型（signed char）	8	单字节	–128～127
整型（int）	16	双字节	–32768～32767
短整型（short int）	16	双字节	–32768～32767
无符号整型（unsigned int）	16	双字节	0～65536
有符号整型（signed int）	16	双字节	–32768～32767
长整型（long int）	32	四字节	–2147483648～2147483647

续表

类　型	长度/bit	长度/Byte	范　围
无符号长整型（unsigned long int）	32	四字节	0～4294967295
有符号长整型（signed long int）	32	四字节	–2147483648～2147483647
单精度浮点型（float）	32	四字节	±1.175e–38～±3.402e+38
双精度浮点型（double）	32	四字节	±1.175e–38～±3.402e+38

📖 说明：

在 CVAVR C 中，若一个表达式中有两个操作数的类型不同，则编译器会自动按以下原则将其转换为同一类型的数据。

（1）如果两个数有一个为浮点型（即单精度或双精度浮点型），则另一个操作数将转换成浮点型。

（2）如果两个数有一个是长整型或无符号长整型，则另一个操作数将转换成相同的类型。

（3）如果两个数有一个是整型或无符号整型，则另一个操作数将转换成相同的类型。

（4）字符型和无符号字符型的优先级最低。

4．常量、变量及存储空间

1）常量　所谓常量就是指在程序运行过程中，其值不能改变的数据。根据数据类型的不同，常量可分为整型常量、字符常量和实数型常量等。

整型常量可以用二进制、八进制、十进制和十六进制数进行表示。表示二进制数时，应在数字的前面加上“0b”的标志，其数码取值只能是“0”和“1”，如“0b10110010”表示二进制的“10110010”，其值为十进制数的 $1\times2^7+1\times2^5+1\times2^4+1\times2^1=178$；表示八进制数时，在数字的前面应加上“O”的标志，其数码取值只能是“0～7”，如“O517”表示八进制的“517”，其值为十进制数的 $5\times8^2+1\times8^1+7\times8^0=335$；表示十六进制时，应在数字的前面加上“0x“或“0X”的标志，其数码取值是数字“0～9”、字母“a～f”或字母“A～F”，如“0x3a”和“0X3A”均表示相同的十六进制数值，其值为十进制数的 $3\times16^1+10\times16^0=58$。

无符号整数常量应在一个数字后面加上“u”或“U”，如 6325U；长整型整数常量应在一个数字后面加上“l”或“L”，如 97L。无符号长整型整数常量应在一个数字后面加上“ul”或“UL”，如 25UL；实数型常量应在一个数字后面加上“f”或“F”，如 3.146F。字符常量是用单引号将字符括起来表示的，如‘a’；字符串常量是用引号将字符括起来表示的，如“CVAVR”。

如果将一个字符串作为一个函数的参数来引用，则这个字符串将自动被当做常量来使用，并且会被放在 Flash 中。例如：

```
//这是一个显示 RAM 中字符串的函数
void   display_ram(char *s){
/* …… */
//这是一个显示 Flash 中字符串的函数
void   display_flash(char flash *s){
/* …… */
void main(){
```

```
display_ram("Hello world");      //不工作，因为寻址的字符串在 Flash 中
display_flash("Hello world");    //工作正常
```

> 注意：
> ①常量可以被定义成数组，最多为 8 维；②由于常量存在 Flash 中，所以必须使用关键词 flash 或 const；③ 常量表达式在编译时会自动求解；④常量可以在函数内部声明。

以下为一些常量的定义：

```
flash  int   intpppeger_constant=2134+87;                  //定义一个“2134+87”的整数
flash  char  char_constant=a;                              //定义一个字符‘a’
flash  long  long_int_constant1=98L;                       //定义一个长整数 98
flash  long  long_int_constant2=0x1234;                    //定义一个十六进制的长整数 0x1234
flash  long  long_int_constant3=0b10011000;                //定义一个二进制的长整数 0b10011000
flash  int   integer_array1[]={3，2，4};                   //定义一个一维数组，其元素分别为
                                                           //3，2，4
flash  int   integer_array2[8]={3，2};                     //定义一个含有 8 个元素的一维数组，
                                                           //其中前 2 个
                                                           //元素为 3，2，其余的元素为 0
flash  int  integer_array3[2，3]={{3，2，1}，{4，5，6}};   //定义了一个 2 行 3 列共 6 个元素的
                                                           //2 维数组
flash  char  string_constant1[]=This is a string constant;  //定义了一个字符串数组
const  char  string_constant2[]=This is also a string constant;  //定义了一个字符串数组
```

2）变量 所谓变量就是在程序运行过程中，其值可以改变的数据。

【局部变量与全局变量】根据实际程序的需求，变量可被声明为局部变量或全局变量。

局部变量是在创建函数时由函数分配的存储器空间，这些变量只能在所声明的函数内使用，而不能被其他的函数访问。但是在多个函数中可以声明变量名相同的局部变量，而不会引起冲突，这是因为编译器会将这些变量视为每个函数的一部分。

全局变量是由编译器分配的存储器空间，可被程序内所有的函数访问。全局变量能够被任何函数修改，并且会保持全局变量的值，以便由其他函数使用。

如果未对全局变量或静态局部变量赋初值，则相当于给该变量赋初值 0，因为 CVAVR 在程序开始时会对全部数据内存清零。在 CVAVR 中，若没有对一个局部变量赋初值，则该变量的值是不确定的。

定义局部变量与全局变量的语法格式如下：

```
[<存储模式>] <类型定义> <标志符>;
```

例如：

```
/*全局变量*/
char   a1;
int    a2;
/*赋初值*/
long   a3=123456;
void   main(void) {
/*局部变量*/
```

```
char   a4;
int    a5;
/*赋初值*/
long   a6=21346541;
```

变量也可以被定义成数组，且最多为 8 维，第一个数组元素编号为 0。如果全局变量数组没有赋初值，则在程序开始时会被自动赋值为 0。例如：

```
int   global_array1[32];                    //定义了一个含 32 个元素的整型数组全局变量，所有
                                            //元素自动赋值为 0
int   global_array2[]={2，2，3};            //定义了一个整型数组全局变量，数组变量元素初值
                                            //分别为 2，2，3
int   global_array3[4]={4，2，3，1};        //定义了一个含 4 个元素的整型数组全局变量，元素
                                            //初值分别为 4，2，3，1
char global_array4[]=This is a string       //定义了一个字符数组全局变量
int   global_array5[32]={1，2，3};          //定义了一个含 32 个元素的整型数组全局变量，前 3
                                            //个元素赋初值，其余 29 个自动赋值为 0
int   multidim_array[2，3]={{1，2，3}，{4，5，6}};   //定义了一个 2 行 3 列共 6 个元素的 2 维数
                                                     //组全局变量
void main(void) {
int   local_array1[10];                     //定义了一个含 10 个元素的整型数组局部变量，每个
                                            //元素的初值为 0
int   local_array2[3]={11，22，33};         //定义了一个含 3 个元素的整型数组局部变量，元素
                                            //初值分别为 11，22，33
char local_array3[7]="Hello";               //定义了一个含 7 个元素的字符串数组局部变量
```

对一些需要被不同的函数调用，并且必须保存其值的局部变量而言，最好将这些局部变量声明为静态变量（static）。如果静态变量没有赋初值，在程序开始时会被自动赋值为 0。例如：

```
int      alfa(void) {
static   int n=1;                           //声明为静态变量
return   n++;
void     main(void) {
int      i;
i=alfa();                                   //返回值为 1
i=alfa();                                   //返回值为 2
```

如果变量在其他的文件中声明，则必须使用关键字 extern。例如：

```
extern   int   xyz;
#include <file_xyz.h> //包含声明 xyz 的文件
```

如果某个变量需占用寄存器，则必须使用 register 关键词告诉编译器分配寄存器给该变量。例如：

```
register   int   abc;   //不管整型变量 abc 是否使用，编译器均会分配一个寄存器给变量 abc
```

为了防止把一个变量分配到寄存器，必须使用 volatile 关键词，并且通知编译器这个变量的赋值受外部变化的支配。所有未被分配到寄存器的全局变量存放在 SRAM 的全局变量

区；所有未被分配到寄存器的局部变量动态地存放在 SRAM 的数据堆栈区。例如：

```
volatile   int   abc;
```

【指定全局变量保存在 SRAM 的地址单元】在程序中可以使用“@”操作符指定全局变量保存在 SRAM 的地址单元。例如：

```
int   a @0x80;                  //指定整型变量 a 保存在 SRAM 的 0x80h 地址单元
struct   b {
              int a;
              char c;
            }
alfa @0x100;                    //结构体 alfa  保存在 SRAM 的 0x100h 地址单元
```

【位变量】位变量是存储在寄存器 R2～R15 的特殊全局变量，用关键词 bit 声明。其语法格式如下：

```
bit <标志符>;
```

例如：

```
/*声明和赋初值*/
bit   alfa=1;                   //定义位变量 alfa，其初值等于 1，存储在 R2 寄存器的 bit0 中
bit   beta;                     //定义位变量 beta，存储在 R2 寄存器的 bit1 中
void main(void)
if   (alfa)
beta=!beta;
/* ..... */
```

根据声明的顺序，位变量的分配从寄存器 R2 的 bit0 开始，按升序排列。由于 R2～R15 中每个寄存器均有 8 位，所以最多可声明 112 个位变量。

若在 CVAVR 编译器中执行菜单命令“Project”→“Configure”，并在“Configure Project”对话框的“Code Generation”选项卡中选中“bit Variables Size”选项，即可给位变量分配寄存器空间。为了给其他的全局变量分配寄存器，给位变量分配的空间要尽可能小一些。

如果位变量没有赋初值，则在程序开始时会自动赋值为 0。在表达式赋值中，位变量会自动转变为无符号字符型。

【给变量分配寄存器】为了给其他的全局变量分配寄存器，给位变量分配的空间要尽可能小。若在 CVAVR 编译器中执行菜单命令“Project”→“Configure”，并在“Configure Project”对话框的“Code Generation”选项卡中选择“Automatic Global Register Allocation”复选框，或者在程序中使用#pragma regalloc+编译指示，则 CVAVR 编译器会自动将位变量未使用完所剩下的 R2～R15 中相关的寄存器分配给字符型和整形全局变量，直到 R15 也分配完为止。

如果自动寄存器分配被禁止，则可以用关键字“register”指定将某个全局变量分配到相应寄存器中。例如：

```
#pragma   regalloc-             //禁止自动寄存器分配
register   int   alfa;          //分配变量 alfa 到一个寄存器中
register   int   beta @14;      //分配变量 beta 到寄存器 R14 和 R15 中
```

3）存储空间　AVR 单片机采用 Harvard（哈佛）结构，CPU 中拥有 3 个独立的空间，即 Flash 存储器、数据存储器和 E^2PROM 存储器。由于 C 语言是针对冯·诺依曼结构的处理器开发的，并不适合描述单片机这 3 种不同的存储空间，所以 AVR 的 C 编译器均对此做了相应的扩充。CVAVR 编译器对每个空间都做了相应的扩充，并引入了 flash 和 eeprom 两个关键词。

【Flash 存储器】Flash 存储器单元的首地址为 0x0000，其容量由 AVR 单片机的型号决定，ATmega16 的 Flash 存储空间为 16KB。Flash 存储器是非易失性存储器，当电源被切断后，保存在其中的数据也不会丢失。

CVAVR 编译器对字符串的处理方式与 ICCAVR 相同，即用户没有指定只保存在 Flash 中的字符串，在启动时将字符串从 Flash 存储器区复制到数据存储区。例如：

```
char   *ptr_to_ram=”this string is placed in RAMI”;        //没有指定，字符串只存储在 Flash 中，因此
                                                          //启动时会将该字符串复制到数据存储空间
char   flash   *ptr_to_ram=”this string is placed in flash”;      //使用了 flash 关键词，字符串只存储
                                                                  //在 Flash 中
```

CVAVR 编译器最多支持 8 个元素的字符串数组，若字符串数组超过 8 个元素，CVAVR 编译器将会报错。

> 注意：
> 这个限制仅针对字符串数组而言，对普通数组没有这个限制。

例如：

```
char   flash   *string[8]={‘I’，‘l’，‘o’，‘v’，‘e’，‘y’，‘o’，‘u’};
```

【数据存储器】AVR 单片机的数据存储器一般包括 3 个相互独立的读/写存储区。全部共有 1120 个数据存储器地址，前 96 个地址为 32 个通用工作寄存器（地址空间为 0x0000～0x001F）和 64 个 I/O 寄存器（地址空间为 0x0020～0x005F），剩下的就是内部 SRAM（地址空间为 0x0060～0x045F）。

通用工作存储器一般用于存储程序运行过程中的局部变量和其他暂时性的数据，甚至还可以用于存储全局变量。64 个 I/O 寄存器则用做微控制器上 I/O 设备的外设接口，而内部 SRAM 则用做通用的变量存储空间，同时也是处理器的堆栈。

【E^2PROM 存储器】存储器中的 E^2PROM 区域是一块可以读/写非易失性存储空间的区域，它常用于存储那些在掉电后不能丢失的和微控制器要反复运用的数据。E^2PROM 的首地址为 0x000，最大的地址取决于所用的单片机型号。

虽然 E^2PROM 存储器是可读写的，但很少用它来存放一般的变量，这是因为 E^2PROM 的写速度非常慢，它要用 1ms 才能完成 1B 数据的写操作。大量使用这类存储器来存放变量会在很大程度上降低处理器的速度。

在 CVAVR 中，可以使用“eeprom”关键字将全局变量分配至 E^2PROM 区域。例如：

```
/*对芯片编程时将“1”写入 E2PROM */
eeprom int alfa=1;
eeprom char beta;
```

```
eeprom long array1[5];
/*对芯片编程时将字符串写入 E2PROM */
eeprom char string[]="Hello";
void main(void) {
int i;
int eeprom *ptr_to_eeprom;          //指向 E2PROM 的指针
alfa=0x85;                          //将 0x85 直接写入 E2PROM
/*或使用指针间接写入*/
ptr_to_eeprom=&alfa;
*ptr_to_eeprom=0x85;
i=alfa;                             //直接从 E2PROM 中读出/
/*或使用指针间接读出*/
i=*ptr_to_eeprom;
}
```



任务 2　运算符和表达式

运算符是告诉编译程序执行特定算术或逻辑操作的符号。C 语言的运算符和表达式相当丰富，这在高级语言中是很少见的。C 语言常用的运算符见表 3-3。

表 3-3　C 语言常用的运算符

名　称	符　号	名　称	符　号
算术运算符	+ - * / % ++ --	条件运算符	?:
赋值运算符	= += –= *= /= %= &= \|= ^= ~= >>= <<=	逗号运算符	,
关系运算符	> < == >= <= !=	指针运算符	* &
逻辑运算符	&& \|\| !	求字节运算符	sizeof
位操作运算符	<< >> ~ ^ \| &	特殊运算符	() []

C 语言规定了一些运算符的优先级和结合性。优先级是指当运算对象两侧都有运算符时，执行运算的先后次序；结合性是指当一个运算两侧的运算符的优先级别相同时的运算顺序。C 语言运算符的优先级及结合性见表 3-4。

表 3-4　C 语言运算符的优先级及结合性

优　先　级	运　算　符	结　合　性
1（最高级）	() [] ,	从左至右
2	! ~ *（指针运算符） &（指针运算符） ++ --	从右至左
3	*（算术运算符） / %	从左至右
4	+ –	从左至右
5	<< >>	从左至右
6	> < >= <=	从左至右
7	== !=	从左至右
8	&（位操作运算符）	从左至右
9	^	从左至右

续表

优　先　级	运　算　符	结　合　性
10	\|	从左至右
11	&&	从左至右
12	\|\|	从左至右
13	?:	从右至左
14 （最低级）	= += –= *= /= %= &= \|= ^= ~= >>= <<=	从右至左

1．算术运算符

算术运算符可用于各类数值运算，包括加（+）、减（−）、乘（*）、除（/）、求余（又称为取模运算，%）、自增（++）和自减（−−）共 7 种运算。

用算术运算符和括号将运算对象连接起来的式子称为算术表达式，其运算对象包括常量、变量、函数和结构等。例如：

```
a+b;
a+b–c;
a*(b+c)– (d–e)/f;
a+b/c–3.6+'b';
```

算术运算符的优先级规定为：先乘除求余，后加减，括号最优先。也就是说，在算术运算符中，乘、除、求余运算符的优先级相同，并高于加减运算符。在表达式中若出现括号，括号中的内容的优先级最高。例如：

```
a–b/c       //在这个表达式中，除号的优先级高于减号，因此先运算 b/c 求得商，再用 a 减
            //去该商
(a+b)*(c–d%e)–f;  //在这个表达式中，括号的优先级最高，因此先运算(a+b)和(c–d%e)，然后再将这
            //两者相乘，最后再减去 f。注意，执行(c–d%e)时，先将 d 除以 e 所得的余数作
            //为被减数，然后用 c 减去该被减数即可
```

算术运算符的结合性规定为自左至右方向，又称为“左结合性”，即当一个运算对象两侧的算术运算符优先级别相同时，运算对象与左面的运算符结合。例如：

```
a–b+c;      //式中 b 两侧的“–”、“+”运算符的优先级别相同，则按左结合性，先执行 a–b 再与 c 相加
a*b/c;      //式中 b 两侧的“*”、“/”运算符的优先级别相同，则按左结合性，先执行 a*b 再除以 c
```

自增（++）和自减（––）运算符的作用是使变量的值增加或减少 1。例如：

```
++a;        //先使 a 的值加上 1，然后再使用 a 的值
a++;        //先使用 a 的当前值进行运算，然后再使 a 加上 1
––a;        //先使 a 的值减少 1，然后再使用 a 的值
a––;        //先使用 a 的当前值进行运算，然后再使 a 减少 1
```

2．赋值运算符和赋值表达式

【一般赋值运算符】在 C 语言中，最常见的赋值运算符为“=”，其作用是计算表达式的值，再将计算结果赋值给左边的变量。赋值运算符具有右结合性，即当一个运算对象两侧的运算符优先级别相同时，运算对象与右面的运算符结合。

其一般格式为

变量=表达式

例如:

```
x=a+b;                  //变量 x 输出为 a 加上 b
s=sqrt(a)+sin(b);       //变量 s 输出的内容为 a 的平方根加上 b 的正弦值
y=i++;                  //变量 y 输出的内容为 i，然后 i 的内容加 1
y=z=x=3;                //可理解为 y=(z=(x=3))
```

如果赋值运算符两边的数据类型不相同，系统将自动进行类型转换，即把赋值号右边的类型转换成左边的类型。具体规定如下：①将实数型转换为整型时，舍去小数部分；②将整型转换为实数型时，数值不变，但将以实数型形式存放，即增加小数部分（小数部分的值为 0）；③将字符型转换为整型时，由于字符型为一个字节，而整型为两个字节，所以将字符的 ASCII 码值放到整型量的低 8 位中，高 8 位为 0；④将整型转换为字符型时，只把低 8 位转换给字符变量。

【复合赋值运算符】在赋值运算符“=”的前面加上其他运算符，就可构成复合赋值运算符。C 语言中的复合赋值运算符包括加法赋值运算符（+=）、减法赋值运算符（–=）、乘法赋值运算符（*=）、除法赋值运算符（/=）、求余（取模）赋值运算符（%=）、逻辑“与”赋值运算符（&=）、逻辑“或”赋值运算符（|=）、逻辑“异或”赋值运算符（^=）、逻辑“取反”赋值运算符（~=）、逻辑“左移”赋值运算符（<<=）、逻辑“右移”赋值运算符（>>=）共 11 种运算。

复合赋值运算首先对变量进行某种运算，然后再将运算的结果赋给该变量。采用复合赋值运算可以简化程序，同时提高 C 程序的编译效率。复合赋值运算表达式的一般格式为

变量　复合赋值运算符　表达式

例如:

```
a+=b;                   //相当于 a=a+b
a–=b;                   //相当于 a=a–b
a*=b;                   //相当于 a=a*b
a/=b;                   //相当于 a=a/b
a%=b;                   //相当于 a=a%b
```

3．关系运算符

在程序中，有时需要先对某些量的大小进行比较，然后根据比较的结果进行相应的操作。在 C 语言中，关系运算符专用于两个量的大小比较，其比较运算的结果只有“真”和“假”两个值。

C 语言中的关系运算符包括大于（>）、小于（<）、等于（==）、大于或等于（>=）、小于或等于（<=）、不等于（!=）共 6 种。

用关系运算符将两个表达式连接起来的式子称为关系表达式。关系运算符两边的运算对象可以是 C 语言中任意合法的表达式或变量。关系表达式的一般格式为

表达式　关系运算符　表达式

关系运算符的优先级别如下：①大于（>）、小于（<）、大于或等于（>=）、小于或等于（<=）属于同一优先级，等于（==）、不等于（!=）属于同一优先级，其中前 4 种运算符的优先级高于后两种运算符的优先级；②关系运算符的优先级别低于算术运算符，但高于赋值运算符的优先级。例如：

```
x>y;                    //判断 x 是否大于 y
a+b<c;                  //判断 a 加上 b 的和是否小于 c
a+b-c= =m*n;            //判断 a 加上 b 的和再减去 c 的差值是否等于 m 乘上 n 的积
```

关系运算符的结合性为左结合。C 语言不像其他高级语言那样有专门的“逻辑值”，它用整数“0”和“1”来描述关系表达式的运算结果，规定用“0”表示逻辑“假”，即当表达式不成立时，运算结果为“0”；用“1”表示逻辑“真”，即当表达式成立时，运算结果为“1”。例如：

```
unsigned  char  x=8,y=9,z=18; //定义无符号字符 x、y、z，它们的初始值分别为 8、9、18
x>y;                    //x=8，y=9，x 小于 y，因此表达式不成立，运算结果为“0”
x+y<z;                  //x 加 y 等于 17，小于 z（z=18），因此表达式成立，运算结果为“1”
(y=18)= =z;             //y 重新赋值为 18 后等于 z（z=18），因此表达式成立，运算结果为“1”
x+6!=z;                 //x 加 6 等于 14，不等于 z（z=18），因此表达式成立，运算结果为“1”
a= =x<y<z               //由于关系运算符的结合性为左结合，所以 x<y 的值为 1，而 1<z 的值
                        //为 1，所以 a 的值为 1
```

4．逻辑运算符

逻辑关系主要包括逻辑“与”、逻辑“或”、逻辑“非”这 3 种基本运算。在 C 语言中，用“&&”表示逻辑“与”运算；用“||”表示逻辑“或”运算；用“!”表示逻辑“非”运算。其中，“&&”和“||”是双目运算符，它要求有两个操作数，而“!”是单目运算符，只要求有一个操作数即可。

> 注意：
> “&”和“|”是位运算符，不要将逻辑运算符与位运算符混淆。

用逻辑运算符将关系表达式或逻辑量连接起来的式子称为逻辑表达式。逻辑表达式的一般格式为

```
表达式    逻辑运算符    表达式
```

逻辑表达式的值是一个逻辑量，为“真”（即“1”）或“假”（即“0”）。对于逻辑“与”运算（&&）而言，参与运算的两个量都为“真”时，结果才为“真”，否则为“假”；对于逻辑“或”运算（||）而言，参与运算的两个量中只要有一个量为“真”时，结果为“真”，否则为“假”；对于逻辑“非”运算（!）而言，参与运算量为“真”时，结果为“假”，参与运算量为“假”时，结果为“真”。

逻辑运算符的优先级别如下：①在 3 个逻辑运算符中，逻辑“非”运算符（!）的优先级最高，其次是逻辑“与”运算符（&&）的优先级，逻辑“或”运算符（||）的优先级最低。②与算术运算符、关系运算符及赋值运算符的优先级相比，逻辑“非”运算符（!）的优先级高于算术运算符的优先级，算术运算符的优先级高于关系运算符的优先级，关系运算符的优先级高于逻辑“与”运算符（&&）和逻辑“或”运算符（||）的优先级，而赋值运算

符的优先级最低。例如：

```
unsigned  char  a=5，b=8，y;    //定义无符号字符 a、b、y，a 的初始值为 5，b 的初始值为 8
y=!a;           //y 的值为逻辑“假”，因为 a=5 为逻辑“真”，所以“!a”为逻辑“假”
y=a||b;         //y 的值为逻辑“真”，因为 a、b 为逻辑“真”，所以“a||b”为逻辑“真”
y=a&&b;         // y 的值为逻辑“真”，因为 a、b 为逻辑“真”，所以“a&&b”为逻辑“真”
y=!a&&b;        //y 的值为逻辑“假”，因为“!”的优先级高于“&&”，需先执行“!a”，其值为
                //逻辑“假”(即“0”)；而“0&&b”的运算为逻辑“假”，所以结果为逻辑“假”
```

5．位操作运算符

能对运算对象进行位操作是 C 语言的一大特点，正是由于有这一特点，使 C 语言具有汇编语言的一些功能，从而使它能对计算机的硬件直接进行操作。

位操作运算符是指按位对变量进行运算，并不改变参与运算的变量的值。如果希望按位改变运算变量的值，则应利用相应的赋值运算。另外，位运算符只能对整型或字符型数据进行操作，不能用于对浮点型数据进行操作。

C 语言中的位操作运算符包括按位“与”（&）、按位“或”（|）、按位“异或”（^）、按位“取反”（～）、按位“左移”（<<）、按位“右移”（>>）共 6 种运算。除了按位“取反”运算符外，其余 5 种位操作运算符都是双目运算符，即要求运算符两侧各有一个运算对象。

【按位“与”（&）】按位“与”的运算规则是参加运算的两个运算对象，若二者相应的位均为“1”，则该位的结果为“1”，否则为“0”。

例如：若 a=0x62=0b01100010，b=0x3c=0b00111100，则表达式 c=a&b 的值为 0x20，即

```
    a:      01100010      (0x62)
    b:  &   00111100      (0x3c)
    ------------------------------
    c   =   00100000      (0x20)
```

【按位“或”（|）】按位“或”的运算规则是参加运算的两个运算对象，若二者相应的位中只要有一位为“1”，则该位的结果为“1”，否则为“0”。

例如：若 a=0xa5=0b10100101，b=0x29=0b00101001，则表达式 c=a | b 的值为 0xad，即

```
    a:      10100101      (0xa5)
    b:  |   00101001      (0x29)
    ------------------------------
    c   =   10101101      (0xad)
```

【按位“异或”（^）】按位“异或”的运算规则是参加运算的两个运算对象，若二者相应的位值相同，则该位的结果为“0”；若二者相应的位值相异，则该位的结果为“1”。

例如：若 a=0xb6=0b10110110，b=0x58=0b01011000，则表达式 c=a^b 的值为 0xee，即

```
    a:      10110110      (0xb6)
    b:  ^   01011000      (0x58)
    ------------------------------
    c   =   11101110      (0xee)
```

【按位“取反”（～）】按位“取反”（ ~ ）是单目运算，用于对一个二进制数按位进行“取反”操作，即“0”变“1”，“1”变“0”。

例如：若 a=0x72=0b01110010，则表达式 a=~a 的值为 0x8d，即

```
    a:      01110010      (0x72)
            ^
    ------------------------------
    a   =   10001101      (0x8d)
```

【按位“左移”（<<）、按位“右移”（>>）】按位“左移”（<<）用于将一个操作数的各二进制位的全部左移若干位，移位后，空白位补“0”，而溢出的位舍弃。

例如：若 a=0x8b=0b10001011，则表达式 a=a<<2 表示将 a 值左移 2 位，其结果为 0x2c，即

```
a:          10001011   （0x8b）
    <<2   10 00101100
a     =     00101100   （0x2c）
```

按位“右移”（>>）用于将一个操作数的各二进制位的全部右移若干位，移位后，空白位补“0”，而溢出的位舍弃。

例如：若 a=0x8b=0b10001011，则表达式 a=a>>2 表示将 a 值右移 2 位，其结果为 0x22，即

```
a:          10001011        （0x8b）
    >>2     00100010 11
a     =     00100010        （0x22）
```

在 ICCAVR 中不能直接访问寄存器的某一位，因此对寄存器进行位操作时，需使用 C 语言中的位运算功能。在 CodeVisionAVR 中可以直接或间接操作寄存器的某一位，它可以直接对位进行置“1”或清“0”操作。

☺ 将寄存器的某一位清零，在 ICCAVR 中可使用按位“与”（&）来实现；在 CodeVisionAVR 中除了可以使用按位“与”（&）外，还可以直接将该位赋为“0”。

例如：将 PC2 清零，而其他位不变，以下 3 条指令均可实现该要求。

```
PORTC &=0xfb;        //相当于 PORTC=PORTC & 0xfb，由于 0xfb 的二进制代码为“11111011”，
                     //所以执行该指令后，PC2 被清零了
PORTC &=~(1<<2);     //相当于 PORTC=PORTC & ~(1<<2)，该指令先执行“(1<<2)”将“1”左
                     //移 2 位，再执行“~”取“反”操作，然后将取“反”的结果与 PORTC
                     //按位“与”操作
PORTC.2=0;           //此指令只能在 CodeVisionAVR 中使用，直接将 PC2 这一位清零
```

☺ 将寄存器的某一位置“1”，在 ICCAVR 中可使用按位“或”（|）来实现；在 CodeVisionAVR 中除了可以使用按位“或”（|）外，还可以直接将该位赋为“1”。

例如：将 PC4 置“1”，而其他位不变，以下 3 条指令均可实现该要求。

```
PORTC |=0x10;        //相当于 PORTC=PORTC | 0x10，由于 0x10 的二进制代码为“00010000”，
                     //所以执行该指令后，PC4 被置 1
PORTC |=(1<<4);      //相当于 PORTC=PORTC & (1<<4)，该指令先执行“(1<<4)”将“1”左移 4
                     //位，再与 PORTC 按位“或”操作
PORTC.4=1;           //此指令只能在 CodeVisionAVR 中使用，直接将 PC4 这一位置“1”
```

☺ 将寄存器的某一位“取反”，在 ICCAVR 中可使用按位“异或”（^）来实现；在 CodeVisionAVR 中除了可以使用按位“异或”（^）外，还可以直接将该位“取反”（~）。

例如：将 PC6 位“取反”，而其他位不变，以下 3 条指令均可实现该要求。

```
PORTC ^=0x40;        //相当于 PORTC=PORTC ^ 0x40，由于 0x40 的二进制代码为“01000000”，
                     //所以执行该指令后，PC4 被“取反”
```

```
PORTC ^=(1<<6);      //相当于 PORTC=PORTC ^ (1<<6)，该指令先执行“(1<<6)”将“1”左移
                     //6 位，再与 PORTC 按位“异或”操作
PORTC.6=～PORTC  //此指令只能在 CodeVisionAVR 中使用，直接将 PC6 这一位“取反”
```

6．条件运算符

条件运算符是 C 语言中唯一的一个三目运算符，它要求有 3 个运算对象，用它可以将 3 个表达式连接起来构成一个条件表达式。条件表达式的一般格式为

```
表达式 1？ 表达式 2 ： 表达式 3
```

条件表达式的功能是首先计算表达式 1 的逻辑值，当逻辑值为“真”时，将表达式 2 的值当做整个条件表达式的值；当逻辑值为“假”时，将表达式 3 的值当做整个条件表达式的值。例如：

```
min=(a<b) ? a : b        //当 a 小于 b 时，min=a；当 a 小于 b 不成立时，min=b
```

7．逗号运算符

逗号运算符又称为顺序示值运算符。在 C 语言中，逗号运算符用于将两个或多个表达式连接起来。逗号表达式的一般格式为

```
表达式 1，表达式 2，…，表达式 n
```

逗号表达式的运算过程是，先求解表达式 1，再求解表达式 2，……依次求解到表达式 *n*。例如：

```
a=2+3，a*8          //先求解 a=2+3，得 a 的值为 5，然后求解 a×8 得 40，整个逗号表达式
                    //的值为 40
a=4*5，a+10，a/6    //先求解 a=4×5，得 20，再求解 a+10 得 30，最后求解 a/6 得 5，整个
                    //逗号表达式的值为 5
```

8．求字节运算符

在 C 语言中提供了一种用于求取数据类型、变量及表达式的字节数的运算符 sizeof。求字节运算符的一般格式为

```
sizeof(表达式) 或 sizeof(数据类型)
```

> 注意：
> sizeof 是一种特殊的运算符，而不是一个函数。通常，字节数的计算在程序编译时就完成了，而不是在程序执行的过程中计算出来的。

9．指针和地址运算符

指针是 C 语言中一个十分重要的概念，CVAVR 中专门规定了一种指针类型的数据。变量的指针就是该变量的地址，也可以说是一个指向某个变量的指针变量。CVAVR 中提供了两个专门用于指针和地址的运算符，即*（取内容）和&（取地址）。

取内容和取地址的一般形式：

变量=*指针变量

指针变量=&目标变量

取内容运算是将指针变量所指向的目标变量的值赋给左边的变量；取地址运算是将目标变量的地址赋给左边的变量。

任务 3　程序结构及流程控制

C 语言是一种结构化编程语言，用户可采用结构化方式编写相关源程序。采用结构化设计的程序具有结构清晰，层次分明，易于阅读修改和维护等特点。

结构化程序由若干个模块组成，每个模块中包含若干个基本结构，而每个基本结构中可有若干条语句。在 C 语言中，有 3 种基本结构，即顺序结构、选择结构和循环结构。

1．顺序结构

顺序结构是一种最基本、最简单的编程结构。在这种结构中，程序由低地址向高地址顺序执行指令代码。如图 3-2 所示，程序要先执行 A，然后再执行 B，两者是顺序执行的关系。顺序结构的特点如下所述。

☺ 执行过程是按顺序从第 1 条语句执行到最后 1 条语句；

☺ 在程序执行过程中，顺序结构程序中的任何一个可执行语句都要运行一次，而且只能运行一次。

2．选择结构

选择结构是对给定的条件进行判断，再根据判断的结果决定执行哪一个分支。如图 3-3 所示，图中 P 代表一个条件，当 P 条件成立（或称为“真”）时，执行 A；否则执行 B。

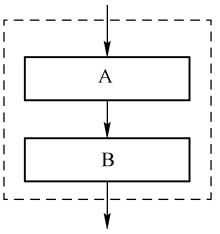


图 3-2　顺序结构

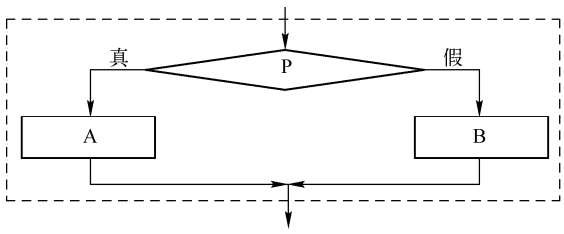


图 3-3　选择结构

📖 注意：

只能执行 A 或 B 之一，两条路径汇合在一起，然后从一个出口退出。

选择结构的语句有 if 条件语句和 switch 开关语句等。

1）if 语句

（1）if 语句的结构形式：if 语句是 C 语言中的一个基本判断语句，它的 3 种结构形式语句如下所述。

【形式 1】

```
if(表达式)
{语句};
```

在这种结构形式中，如果括号中的表达式成立，则程序执行“{}”中的语句；否则程序将跳过“{}”中的语句部分，顺序执行其他语句。例如：

```
if   (PINB.0= =0)
{                                    //如果 PB.0 端口为低电平，则执行下述语句
      PORTB.4=～PORTB.4;             //PB.4 端口输出相反的状态
      PORTB.5=0;                     //PB.5 端口输出为低电平
}
```

【形式 2】

```
if (表达式)
   {语句 1;}
else
   {语句 2;}
```

在这种结构形式中，如果括号中的表达式成立，则程序执行“{语句 1;}”中的语句；否则程序执行“{语句 2;}”中的语句。例如：

```
if   (PINB.0= =0)
    {                                //如果 PB0 端口为低电平，则执行下述语句
      PORTB.4=～PORTB.4;             //PB4 端口输出相反的状态
      PORTB.5=0;                     //PB5 端口输出为低电平
    }
else
    {                                //如果 PB0 端口不是低电平，则执行下述语句
      PORTB.7=～PORTB.7;             //PB7 端口输出相反的状态
      PORTB.5=1;                     //PB5 端口输出为高电平
}
```

【形式 3】

```
if (表达式 1)
{语句 1;}
else if (表达式 2)
   {语句 2;}
else if (表达式 3)
   {语句 3;}
……
else if (表达式 m)
   {语句 m;}
else
   {语句 n;}
```

在这种结构形式中，如果括号中的表达式 1 成立，则程序先执行“{语句 1;}”中的语句，然后退出 if 选择语句，不执行下面的语句；如果表达式 2 成立，则程序先执行“{语句 2;}”中的语句，然后退出 if 选择语句，不执行下面的语句；如果表达式 3 成立，则程序先

执行“{语句 3;}”中的语句，然后退出 if 选择语句，不执行下面的语句；……如果表达式 *m* 成立，则程序先执行“{语句 *m*;}”中的语句，然后退出 if 选择语句，不执行下面的语句；如果上述表达式均不成立，则程序执行“{语句 *n*;}”中的语句。

例如，根据 a 值的大小决定 numb 系数，编写的程序段如下所示。

```
if   (a>6500)
   {numb=1;}
else if   (a>6000)
   {numb=0.8;}
else if   (a>5800)
   {numb=0.6;}
else if   (a>5600)
   {numb=0.4}
else
   {numb=0;}
```

（2）if 语句的嵌套：如果 if 语句中又包含一个或多个 if 语句时，这种情况称为 if 语句的嵌套。if 语句的嵌套基本形式如下：

```
外层嵌套 if 语句 { if( 表达式 1)
                   {
                     if( 表达式 2)
                       { 语句 1:}      } 内层嵌套 if 语句
                     else
                       { 语句 2:}
                   }
                 else
                   {
                     if( 表达式 3)
                       { 语句 3:}      } 内层嵌套 if 语句
                     else
                       { 语句 4:}
                   }
```

【例 3-1】本例以 ATmega16 单片机为例说明 if 条件语句，其 LED 控制电路原理图如图 3-4 所示。在 ATmega16 单片机的 PC0 和 PC4 端口分别接 D1 和 D2 这两个 LED，该实例的控制任务是当开关 K1 闭合时，D1 点亮，D2 熄灭；当开关 K1 断开时，D1 熄灭，而 D2 点亮。编写的程序如下：

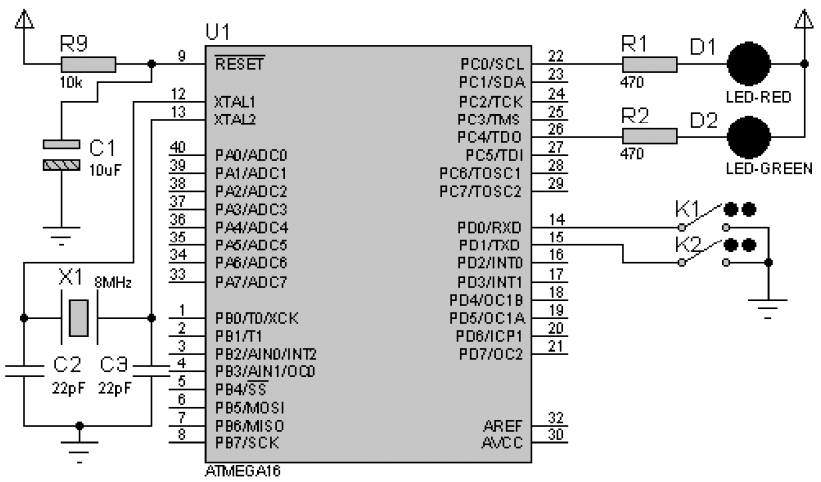


图 3-4　LED 控制电路原理图

```
/*************************************************
 File name:          if 语句应用.C
 Chip type:          ATmega16
 Clock frequency:    8.0MHz
*************************************************/
#include "mega16.h"
#define uchar unsigned char
//定义开关及 LED 与端口的连接
#define   redLED          PORTC.0
#define   greenLED        PORTC.4
void main(void)
{    DDRC=0xFF;
     PORTC=0xFF;
     DDRD=0x00;
     PORTD=0xff;
     while(1)
      {
        if (PIND & 0x01)                    //检测 PD0 端口上的开关为断开状态
          {
             redLED=0x1;                    //D1 熄灭（低电平有效）
           greenLED=0x0;                    //D2 点亮
          }
        else                                //检测 PD0 端口上的开关为闭合状态
          {
             redLED=0x0;                    //D1 点亮
           greenLED=0x1;                    //D2 熄灭
          }
      }
}
```

2）switch 语句　在实际使用中，通常会碰到多分支选择问题，此时可以使用 if 嵌套语句来实现。但是，如果分支很多，if 语句的层数太多、程序冗长，则可读性会降低，而且很容易出错。基于此，在 C 语言中使用 switch 语句可以很好地解决多重 if 嵌套容易出现的问题。switch 语句是另一种多分支选择语句，用于实现多方向条件分支的语句。

【switch 语句格式】

```
switch (表达式)
    case   常量表达式 1:
             {语句 1;} break;
    case   常量表达式 2:
             {语句 2;} break;
    case   常量表达式 3:
             {语句 3;} break;
             ⋮
    case   常量表达式 m:
             {语句 m;} break;
    default:
             {语句 n;} break;
```

【switch 语句使用说明】

☺ switch 后面括号内的“表达式”既可以是整型表达式或字符型表达式，也可以是枚举型数据。

☺ 当 switch 后面表达式的值与某一“case”后面的常量表达式相等时，就执行该“case”后面的语句，如果后面遇到 break 语句就退出 switch 语句。若所有“case”中常量表达式的值都没有与表达式的值相匹配的，就执行 default 后面的语句。

☺ 每一个 case 的常量表达式的值必须互不相同，否则就会出现互相矛盾的现象（对同一个值，提供了两种或多种解决方案）。

☺ 每个 case 和 default 的出现次序不影响执行结果，可先出现“default”再出现其他的“case”。

☺ 假如在 case 语句的最后没有“break;”，则流程控制将转移到下一个 case 继续执行。因此，在执行完一个 case 分支后，应使流程跳出 switch 结构，即终止 switch 语句的执行，这可使用一个 break 语句来完成。

【例 3-2】本例仍以图 3-4 所示的电路原理图为例说明 switch 的用法。此时的任务是当按下开关 K1 时，D1 亮；当按下开关 K2 时，D2 亮。编写的程序如下：

```
/*************************************************
 File name:            switch 语句应用.C
 Chip type:            ATmega16
 Clock frequency:   8.0MHz
*************************************************/
#include "mega16.h"
#define uchar unsigned char
//定义开关及 LED 与端口的连接
#define   redLED          PORTC.0
#define   greenLED        PORTC.4
void main(void)
{    DDRC=0xFF;
   PORTC=0xFF;
    DDRD=0x00;
   PORTD=0xff;
    while(1)
     { switch (PIND & 0x03)          //该表达式判断 K1 和 K2 是否闭合，如果闭合则与之相连
                                     //的引脚为低电平
       {
         case 0x02:                  //表达式的值等于 0x02，表示 K1 开关处于闭合状态
           redLED=0x0;               //D1 亮
           break;                    //跳出 switch 结构，不执行下面的语句
         case 0x01:                  //表达式的值等于 0x01，表示 K2 开关处于闭合状态
           greenLED =0x0;            //D2 亮
           break;
         default:                    //表达式的值既不等于 0x01，又不等于 0x02 表示两个开关
                                     //均未闭合
           redLED=0x1;               //两个 LED 均不亮
           greenLED=0x1;
```

```
                break;
            }
        }
    }
```

3．循环语句与控制结构

在许多实际问题中，需要程序有规律地重复执行，此时可以采用一些循环语句来实现。在 C 语言中，用于实现循环的语句有 goto 语句、while 语句、do-while 语句和 for 语句等。

1）goto 语句 goto 语句为无条件转向语句，该语句可以实现循环。goto 语句的一般格式如下：

```
goto  语句标号;
```

其中，语句标号不必特殊加以定义，它是一个任意合法的标志符，其命名规则与变量名相同，由字母、数字和下画线组成，并且第一个字符必须为字母或下画线，不能用整数作为标号。当这个标志符加上一个“:”一起出现在函数内某处时，则执行 goto 语句后，程序将跳转到该标号处并执行其后的语句。标号必须与 goto 语句同处于一个函数中，但可以不在一个循环层中。

结构化程序设计主张限制使用 goto 语句，主要是因为它将使程序层次不清，且不易读，但也并不绝对禁止使用 goto 语句。在多层嵌套退出时，用 goto 语句则比较合理。一般来说，使用 goto 语句可以有以下两种用途：与 if 语句一起构成循环结构；从循环体中跳转到循环体外。

【与 if 语句一起构成循环结构】

【例 3-3】用 if 语句和 goto 语句构成循环结构，求 $\sum_{n=0}^{50} n$，编写的程序如下所示。

```
/*************************************************
  File name:          goto 语句应用.C
  Chip type:          ATmega16
  Clock frequency:    8.0MHz
*************************************************/
#include "mega16.h"
void main(void)
   {
     int i=0，sum=0;
     loop: if(i<=50)
        {
          sum=sum+i;
          i++;
          goto loop;
        }
   }
```

该程序的运行结果为 1275（十六进制为 04FB），在 AVR Studio 或 Proteus 中可观察到 R19 中的内容为 04，R18 中的内容为 FB。

【从循环体中跳转到循环体外】在 C 语言中，如果要跳出本层循环和结束本次循环，可

以使用 break 语句和 continue 语句。goto 语句的使用机会已大大减少，只是需从多层循环的内层跳到多层循环体外时才用到 goto 语句。但是这种用法不符合结构化原则，一般不宜采用，只有在特殊情况（如需要大大提高生成代码的效率）时才使用。

2）while 语句　while 语句很早就出现在 C 语言编程的描述中了，它是最基本的控制元素之一，用于实现“当型”循环结构。while 语句的一般格式如下：

```
while (表达式)
  {语句;}
```

若程序的执行进入 while 循环的顶部时，将对表达式求值。如果该表达式为“真”（非零），则执行 while 循环内的语句。当执行到循环底端时，马上返回到 while 循环的顶部，再次对表达式进行求值。如果值仍为“真”，则继续循环，否则完全绕过该循环，而继续执行紧跟在 while 循环后的语句，其流程如图 3-5 所示。

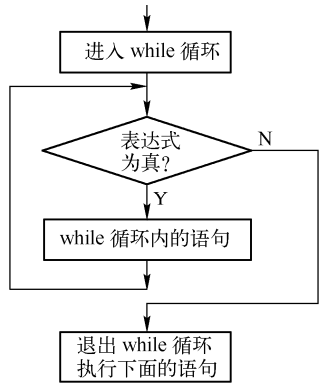


图 3-5　while 语句的流程图

【例 3-4】用 while 语句，求 $\sum_{n=0}^{50} n$，编写的程序如下所示。

```
/*****************************************************
 File name:          while 语句应用.C
 Chip type:          ATmega16
 Clock frequency:    8.0MHz
*****************************************************/
#include "mega16.h"
void main(void)
  {
    int n=0，sum=0;
    while (n<=50)
      {
        sum=sum+n;
        n++;
      }
  }
```

该程序的运行结果为 1275（十六进制为 04FB），在 AVR Studio 或 Proteus 中可观察到 R19 中的内容为 04，R18 中的内容仍为 FB。

3）do-while 语句　do-while 循环与 while 循环十分相似，这两者的区别在于：do-while 语句是先执行循环后判断，即循环内的语句至少执行一次，然后再判断是否继续循环，其流程如图 3-6 所示；while 语句是在每次执行的指令前先判断。do-while 语句的一般格式如下：

```
do
{语句;}
While (条件表达式);
```

【例 3-5】用 do-while 语句，求 $\sum_{n=0}^{50} n$，编写的程序如下所示。

```
/*************************************************
* File name:              do-while 语句应用.C
Chip type:              ATmega16
Clock frequency:    8.0MHz
*************************************************/
#include "mega16.h"
void main(void)
  {
    int n=0，sum=0;
    do
      {
        sum=sum+n;
        n++;
       }
    while (n<=50);
  }
```

4）for 语句 在 C 语言中，for 语句的使用最为灵活，完全可以取代 while 语句或 do-while 语句。它不仅可以用于循环次数已经确定的情况，而且可以用于循环次数不确定而只给出循环结束条件的情况。for 语句的一般格式如下：

```
for (表达式 1;表达式 2;表达式 3)
{语句;}
```

for 语句的流程如图 3-7 所示，其执行过程如下所述。

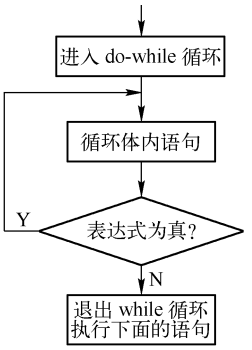


图 3-6 do-while 语句的流程图

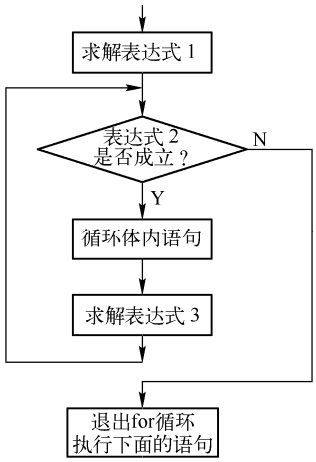


图 3-7 for 语句的流程图

（1）先对表达式 1 赋初值，进行初始化。

（2）判断表达式 2 是否满足给定的循环条件，若满足循环条件，则执行循环体内语句，然后执行第（3）步；若不满足循环条件，则结束循环，转到第（5）步。

（3）若表达式 2 为“真”，则在执行指定的循环语句后，求解表达式 3。

（4）回到第（2）步继续执行。

（5）退出 for 循环，执行下面的语句。

for 语句最简单的应用形式（即最易理解的形式）如下：

```
for (循环变量赋初值;循环条件；循环变量增值)
{语句;}
```

【例 3-6】用 for 语句，求 $\sum_{n=0}^{50} n$，编写的程序如下所示。

```
/*****************************************************
 File name:            while 语句应用.C
 Chip type:            ATmega16
 Clock frequency:   8.0MHz
*****************************************************/
#include "mega16.h"
void main(void)
  {
   int n，sum=0;
   for (n=0;n<=50;n++)
       {
          sum=sum+n;
       }
  }
```

显然，用 for 语句简单、方便。for 语句的一般形式还可以用相应的 while 循环形式来表示：

```
表达式 1;
while (表达式 2)
   {
语句;
     表达式 3;
   }
```

同样，for 语句的一般形式还可以用相应的 do-while 循环形式来表示：

```
表达式 1;
do
   {
语句;
     表达式 3;
   }
while (表达式 2)
```

for 语句使用得最为灵活，除了可以取代 while 语句或 do-while 语句外，还在结构形式上体现了其灵活性。下面对 for 循环语句的 6 种特例进行说明。

（1）for 语句中小括号内的表达式 1 缺省时，应在 for 语句前给循环变量赋初值。

注意：

虽然表达式 1 省略了，但是表达式 1 后面的分号不能省略。例如：

```
int n，sum=0;
for (;n<=50;n++)
{
sum=sum+n;
}
```

该程序段执行时，不对 n 设置初值，直接跳过“求解表达式 1”这一步，而其他不变。

（2）for 语句中小括号内的表达式 2 缺省时，不判断循环条件，默认循环条件始终为“真”，使循环无休止地进行下去。例如：

```
int n，sum=0;
for (n=0; ;n++)
{
sum=sum+n;
}
它相当于：
int n，sum=0;
while (1)
  {
        sum=sum+n;
        n++;
  }
```

（3）for 语句中小括号内的表达式 3 缺省时，在程序中应书写相关语句以保证循环能正常结束。例如：

```
int n，sum=0;
for (n=0;n<=50;)
{
sum=sum+n;
   n++;
}
```

在此程序段中，将 n++的操作不放在 for 语句的表达式 3 的位置处，而作为循环体的一部分，其效果是一样的，都能使循环正常结束。

（4）for 语句中小括号内的表达式 1 和表达式 3 缺省，而只给出循环条件时，在此种情况下，完全等效于 while 语句。例如：

```
int n，sum=0;
for (;n<=50;)
{
sum=sum+n;
   n++;
}
```

它相当于：

```
int n，sum=0;
```

```
while   (n<=50)
{
sum=sum+n;
    n++;
}
```

（5）for 语句中小括号内的 3 个表达式都缺省时，既不设置初值，也不判断条件，而循环变量也不增值，则会使程序无休止地执行循环体。例如：

```
for (; ;)
  {…… /*循环体*/}
它相当于：
while (1)
  {…… /*循环体*/}
```

（6）for 语句中没有循环体。例如：

```
int   n;
 for(n=0;n<1000;n++)
     {;}
```

此例在程序段中起延时作用。

5）break 和 continue 语句

（1）break 语句：break 语句通常可以用在 switch 语句或循环语句中。当 break 语句用于 switch 语句中时，可使程序跳出 switch 而执行 switch 以后的语句；当 break 语句用于 while、do-while、for 循环语句中时，可使程序提前终止循环而执行循环后面的语句，通常 break 语句总是与 if 语句连在一起的，即满足条件时便跳出循环。break 语句的一般格式如下：

```
break;
```

> 注意：
>
> ①break 语句不能用在循环语句和 switch 语句之外的任何其他语句中；②break 语句只能跳出它所处的那一层循环，而不像 goto 语句那样可以直接从最内层循环中跳出来。因此，要退出多重循环时，采用 goto 语句比较方便。

（2）continue 语句：continue 语句一般用在 while、do-while、for 循环语句中，其功能是跳过循环体中剩余的语句而强行执行下一次循环。通常 continue 语句总是与 if 语句连在一起的，用于加速循环。continue 语句的一般格式如下：

```
continue;
```

continue 语句和 break 语句的区别：break 语句结束循环时，不再进行条件判断；continue 语句只能结束本次循环，不终止整个循环。



任务 4　数组与指针

1．数组

数组是一组具有固定数目的相同类型成分分量的有序数据集合。数组是 C 语言提供的

一种最简单的构造类型，其成分分量的类型为该数组的基本类型。整型变量的有序集合称为整型数组，字符型变量的有序集合称为字符型数组。数组中的每个元素都属于同一个数据类型，在同一数组中不允许出现不同类型的变量。

在数组中，可以用一个统一的数组名和下标来唯一地确定数组中的元素。数组中的下标放在方括号中，是从 0 开始（0，1，2，3，4，…，*n*）的一组有序整数。例如，数组 a[*i*]，当 *i*=0，1，2，3，…，*n* 时，a[0]，a[1]，a[2]，…，a[*n*]分别是数组 a[*i*]的元素。数组中有一维、二维、三维和多维数组之分，常用的有一维、二维和字符数组。

1）一维数组 数组只有一个下标时，称为一维数组。在 C 语言中，使用数组前，需先对其进行定义。一维数组的定义方式如下：

```
类型说明符    数组名［常量表达式］;
```

其中，类型说明符是任一种基本数据类型或构造数据类型（如 int，char 等）。数组名是用户定义的数组标志符，即合法的标志符。方括号中的常量表达式表示数组元素的个数，也称为数组的长度。例如：

```
unsigned int a[8];          //定义了含有 8 个元素的无符号整型数组 a
float b[10], c[16];         //定义了含有 10 个元素的实型数组 b，含有 16 个元素的实型数组 c
unsigned char ch[20];       //定义了含有 20 个元素的字符数组 ch
```

📖 注意:

对于数组类型的定义应注意以下 4 点。

☺ 数组名的定义规则和变量名相同，应遵循标志符命名规则。在同一程序中，数组名不能重名，即不能与其他变量名相同。

☺ 数组名后是用方括号括起来的常量表达式，不能使用圆括号。

☺ 方括号中的常量表达式表示数组元素的个数，如 a[10]表示数组 a 有 10 个元素。每个元素由不同的下标表示，在数组中的下标是从 0 开始计算，而不是从 1 开始计算的。因此，a 的 10 个元素分别为 a[0]，a[1]，…，a[9]，而没有 a[10]这个数组元素。

☺ 常量表达式中可以包括常量和符号常量，不能包含变量。也就是说，C 语言中数组元素的个数不能在程序运行过程中根据变量值的不同而随机修改，数组的元素个数在程序编译阶段就已经确定了。

定义一维数组后，就可以引用这个一维数组中的任何元素，且只能逐个引用而不能一次引用整个数组的元素。引用数组元素的一般形式如下：

```
数组名[下标]
```

这种引用数组元素的方法称为“下标法”。C 语言规定，以下标法使用数组元素时，下标可以越界，即下标可以不在 0～（长度−1）的范围内。例如，定义数组为 a[3]，能合法使用的数组元素是 a[0]、a[1]、a[2]，而 a[3]、a[4]虽然也能使用，但由于下标越界，超出数组元素的范围，所以程序运行时，可能会出现不可预料的结果。

例如，对 10 个元素的数组进行赋值时，必须使用循环语句逐个输出各个变量，如下所示。

```
int i，a[10];                //定义变量 i 及含 10 个元素的一维数组 a
for (i=0;i<10;i++)
```

```
  {
a[i]=0;
}
```

而不能类似于下列的方法用一个语句输出整个数组变量。

```
int i，a[10];
a=0;
```

除了用赋值语句对数组元素赋值外，还可以采用初始化赋值和动态赋值的方法给数组赋值。

数组初始化是指在定义数组的同时给数组元素赋值。虽然数组赋值可以在程序运行期间用赋值语句进行赋值，但是这样将耗费大量的运行时间，尤其是对大型数组而言，这种情况更加突出。采用数组初始化的方式赋值时，由于数组初始化是在编译阶段进行的，这样将减少运行时间，提高效率。

一维数组初始化赋值的一般形式如下：

```
类型说明符　数组名[常量表达式]={值，值，值，…，值};
```

其中，在“{}”中的各数据值即为各元素的初值，各值之间用逗号间隔。例如：

```
flash unsigned char tab[8]={0xfe,0xfd,0xfb,0xf7,0xef,0xdf,0xbf,0x7f};
```

经过上述定义的初始化后，各个变量值为：tab[0]=0xfe;tab[1]=0xfd;tab[2]=0xfb;tab[3]=0xf7;tab[4]=0xef;tab[5]=0xdf;tab[6]=0xbf;tab[7]=0x7f。

C 语言对一维数组元素的初始化赋值还有以下特例。

（1）只给一部分元素赋初值：当“{}”中值的个数少于元素个数时，可以只给前面部分元素赋值。例如：

```
flash unsigned char tab[10]={0x00,0x00,0x07,0x02,0x02,0x02,0x7F};
```

在此语句中，定义 tab 数组有 10 个元素，但“{}”内只提供了 7 个初值，这表示只给前面 7 个元素赋值，而后面 3 个元素的初值为 0。

（2）给全部元素赋值相同值时，应在“{}”内将每个值都写上。例如：

```
int a[10]={2,2,2,2,2,2,2,2,2,2};
```

而不能写为

```
int a[10]=2;
```

（3）给全部元素赋值，在数组说明中进行时，可以不给出数组元素的个数。例如：

```
flash unsigned char tab1[24]={0x00,0x00,0x7F,0x1E,0x12,0x02,0x7F,0x00,
                              0x00,0x00,0x07,0x02,0x02,0x02,0x7F,0x00,
                              0x00,0x00,0x7F,0x1E,0x12,0x02,0x7F,0x00};
```

可以写为

```
flash unsigned char tab1[]={0x00,0x00,0x7F,0x1E,0x12,0x02,0x7F,0x00,
```

```
                                    0x00,0x00,0x07,0x02,0x02,0x02,0x7F,0x00,
                                    0x00,0x00,0x7F,0x1E,0x12,0x02,0x7F,0x00};
```

由于数组 tab1 初始化时“{}”内有 24 个数，所以系统自定义 tab1 的数组个数为 24，并将这 24 个字符分配给 24 个数组元素。

2）二维数组 C 语言允许使用多维数组，最简单的多维数组就是二维数组。实际上，二维数组是以一维数组为元素构成的数组。二维数组的定义方式如下：

```
类型说明符    数组名［常量表达式 1］［常量表达式 2］;
```

其中，常量表达式 1 表示第 1 维下标的长度，常量表达式 2 表示第 2 维下标的长度。二维数组存取顺序是：按行存取，先存取第 1 行元素的第 0 列，第 1 列，第 2 列，……直到第 1 行的最后一列；然后返回到第 2 行开始，再存取第 2 行的第 0 列，第 1 列，第 2 列，……直到第 2 行的最后一列。以此类推，直到最后一行的最后一列存完为止。例如：

```
int a[4][6]
```

该列定义了 4 行 6 列共 24 个元素的二维数组 a[][]，其存取顺序如下：

```
→ a[0] [0]→ a[0] [1]→ a[0] [2]→ a[0] [3]→ a[0] [4]→ a[0] [5]┐
└→ a[1] [0]→ a[1] [1]→ a[1] [2]→ a[1] [3]→ a[1] [4]→ a[1] [5]┐
└→ a[2] [0]→ a[2] [1]→ a[2] [2]→ a[2] [3]→ a[2] [4]→ a[2] [5]┐
└→ a[3] [0]→ a[3] [1]→ a[3] [2]→ a[3] [3]→ a[3] [4]→ a[3] [5]
```

二维数组元素引用的一般形式为：

```
数组名［下标］［下标］
```

其中，下标可以是整数，也可以是整数表达式。例如：

```
a[2][4]                  //表示 a 数组第 2 行第 4 列的元素
a[3−1][2*2−1]            //不要写成 a[2, 3]，也不要写成 a[3−1, 2*2−1]的形式
```

在使用数组时，下标值应在已定义的数组大小范围内，以避免越界错误。例如：

```
int   a[3][4];
      ⋮
a[3][4]=4;       //定义 a 为 3×4 的数组，其行下标值最大为 2，列下标值最大为 3，而 a[3][4]超
                 //过数组范围
```

二维数组初始化也是指在类型说明时给各下标变量赋以初值。对二维数组赋值时，可以按以下方法进行。

（1）按行分段赋值：按行分段赋值是指将第 1 个“{}”内的数值赋给第 1 行的元素，将第 2 个“{}”内的数值赋给第 2 行的元素，以此类推。采用这种方法比较直观，例如：

```
flash unsigned char tab[3][4]={ {0x00,0x00,0x7F,0x1E},{0x12,0x02,0x7F,0x00},
                                {0x02,0x02,0x7F,0x00}};
```

（2）按行连续赋值：按行连续赋值是指将所有数据写在一个“{}”内，按数组排列的顺序对各个元素赋初值。例如：

```
flash unsigned char tab[3][4]={0x00,0x00,0x7F,0x1E,0x12,0x02,0x7F,0x00,0x02,
                               0x02,0x7F,0x00};
```

从这段赋值可以看出，第（2）种方法与第（1）种方法完成相同任务，都是定义同一个二维数组 tab 且赋相同的初始值，但是第（2）种方法没有第（1）种方法直观，如果二维数组需要赋的初始值比较多时，采用第（2）种方法将会在“{}”内写一大片，容易遗漏，也不容易检查。

（3）对部分元素赋初值：可以对二维数组的部分元素赋初值，未赋值的元素自动取“0”值。例如：

```
int  a[3][4]={{1},{3},{6}};          //二维数组 a 各元素的值为{{1,0,0,0},{3,0,
                                     //0,0},{6,0,0,0}}
int  b[3][4]={{2},{1,3},{2,4,3}};    //二维数组 b 各元素的值为{{2,0,0,0},{1,3,
                                     //0,0},{2,4,3,0}}
int  c[3][4]={{2},{3,5}};            //二维数组 c 各元素的值为{{2,0,0,0},{3,5,
                                     //0,0},{0,0,0,0}}
int  d[3][4]={{1},{},{2,3,4}};       //二维数组 d 各元素的值为{{1,0,0,0},{0,0,
                                     //0,0},{2,3,4,0}}
```

（4）给元素赋初值时，可以不指定第 1 维的长度：如果对全部元素都赋初始值，则定义数组时对第 1 维的长度可以不指定，但第 2 维的长度不能省略。例如：

```
int  a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

与下面的定义等价：

```
int  a[ ][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

如果只对部分元素赋初始值，则定义数组时对第 1 维的长度可以不指定，但第 2 维的长度不能省略，且应分行赋初始值。例如：

```
int  a[ ][4]={{1,2,3},{},{5}};
```

该程序段定义了 3 行 4 列的二维数组，元素各初始值分别为{{1,2,3,0},{0,0,0,0},{5,0,0,0}}。

3）字符数组　用于存放字符数据的数组称为字符数组。字符数组中的一个元素存放一个字符，因此可以用字符数组来存放长度不同的字符串。

字符数组的定义与前面介绍的类似，即

```
(unsigned) char  数组名［常量表达式］;
```

例如：

```
char  a[10];                         //定义了包含 10 个元素的字符数组 a
```

字符数组也可以是二维或多维数组，和数值型多维数组相同。例如：

```
char  b[3][5];                       //定义了 3 行 5 列共 15 个元素的二维字符数组 b
```

字符数组和数值型数组一样，也允许在定义时进行初始化赋值。例如：

```
unsigned char a[7]={‘p’，‘r’，‘o’，‘t’，‘e’，‘u’，‘s’}  //将 7 个字符分别赋给 a[0]～a[6]这 7 个元素
```

如果“{ }”中提供的初始值个数（即字符个数）大于数组长度，则 C 语言会将其作为语法错误处理。如果初始值个数小于数组长度，则只将这些字符赋给数组中前面的元素，其余的元素自动定义为空字符（即‘\0’）。对全体元素赋初值时，也可以省去长度。例如：

```
unsigned char a[13 ]={‘C’，‘o’，‘d’，‘e’，‘V’，‘i’，‘s’ ，‘i’，‘o’ ，‘n’，‘A’ ，‘V’，‘R’};
```

也可以写成：

```
unsigned char a[ ]= {‘C’，‘o’，‘d’，‘e’，‘V’，‘i’，‘s’ ，‘i’，‘o’ ，‘n’，‘A’ ，‘V’，‘R’};
```

字符串常量是由双引号括起来的一串字符。在 C 语言中，将字符串常量作为字符数组来处理。例如，在上例中就是用一个一维字符数组来存放一个字符串常量“CodeVisionAVR”的，这个字符串的实际长度与数组长度相等。如果字符串的实际长度与数组长度不相等时，为了测定字符串的实际长度，C 语言规定以字符“\0”作为字符串结束标志，也就是说，在遇到第 1 个字符“\0”时，表示字符串结束，由它前面的字符组成字符串。

在 C 语言中没有专门的字符串变量，通常用一个字符数组来存放一个字符串。若将一个字符串存入一个数组时，也将结束符“\0”存入数组，并以此作为该字符串是否结束的标志。

如果将字符串直接给字符数组赋初值时，可采用类似于以下的两种方法：

```
unsigned char a[ ]={“CodeVisionAVR”};
unsigned char a[ ]=“CodeVisionAVR”;
```

2．指针

所谓指针就是指内存中的地址，它可能是变量的地址，也可能是函数的入口地址。如果指针变量存储的地址是变量的地址，则称该指针为变量的指针（或变量指针）；如果指针变量存储的地址是函数的入口地址，则称该指针为函数的指针（或函数指针）。

1）变量指针和指向变量的指针变量 指针变量与变量指针的含义不同。指针变量也简称指针，是指一个变量，且该变量是指针类型的；而变量指针虽然也是指一个变量，且该变量也是指针类型的，但它存放的是另一个变量的地址。

在 C 语言中，所有的变量在使用前必须定义，以确定其类型。指针变量也一样，由于它是用于专门存放地址的，所以必须将它定义为“指针类型”。指针定义的一般形式如下：

```
类型标志符  *指针变量名;
```

其中，类型标志符就是本指针变量所指向的变量的数据类型；“*”表示这是一个指针变量；指针变量名就是指针变量的名称。例如：

```
int    *ap1;                  //定义整型指针变量 ap1
char   *ap2，*ap3;             //定义了两个字符型指针变量 ap2 和 ap3
float  *ap4;                  //定义了实数型指针变量 ap4
```

📖 注意：

☺ 指针变量名前的“*”表示该变量为指针变量，上例中的指针变量名为 ap1、ap2、ap3、ap4，而不是*ap1、*ap2、*ap3、*ap4，这与定义变量有所不同。

☺ 一个指针变量只能指向同一个类型的变量，上例中的 ap1 只能指向整型变量，不能指向字符型或实数型指针变量。

指针变量在使用前也要先定义说明，然后赋予具体的值。指针变量只能赋予地址，而不能赋予任何其他数据，否则将引起错误。在 C 语言中，变量的地址由编译系统分配，用户不知道变量的具体地址。

有两个有关的运算符，即“&”和“*”。其中，“&”为取地址运算符；“*”为指针运算符（或称“间接访问”运算符）。

在 C 语言中，指针变量的引用是通过取地址运算符“&”来实现的。使用取地址运算符“&”和赋值运算符“=”就可以使一个指针变量指向一个变量。

例如，给指针变量 p 所对应的内存地址单元中装入变量 x 所对应的内存单元地址，可使用以下程序段来实现：

```
int   x;              //定义整型变量 x
int   *p=&x;          //指针变量声明时初始化
```

还可以采用以下程序段来实现：

```
int   x;              //定义整型变量 x
int   *p;             //定义整型指针变量 p
p=&x;                 //用赋值语句对指针赋值
```

2）数组指针和指向数组的指针变量　指针既然可以指向变量，当然也可以指向数组。所谓数组指针是指数组的起始地址，数组元素的指针是数组元素的地址。若有一个变量用于存放一个数组的起始地址（指针），则称它为指向数组的指针变量。

定义一个指向数组元素的指针变量的方法与指向变量的指针变量的定义方法相同。例如：

```
int   x[6];           //定义含有 6 个整型数据的数组
int   *p;             //定义指向整型数据的指针 p
p=&x[0];              //对指针 p 赋值，此时数组 x[5]的第 1 个元素 x[0]的地址就赋给了指针变量 p
p=x;                  //对指针 p 赋值，此种引用的方法与“p=&x[0];”的作用完全相同，但形式
                      //上更简单
```

在 C 语言中，数组名代表数组的首地址，也就是第 0 号元素的地址。因此，语句“P=&x[0];”和“P=x;”是等价的。还可以在定义指针变量时赋给初值：

```
int   *p=&x[0];   //或者 int   *p=x;
```

等价于：

```
int   * p ;
p=&x[0] ;
```

如果 p 指向一个一维数组 x[6]，并且已给 p 赋予了一个初值&x[0]，则可以使用以下 3 种方法引用数组元素。

【下标法】C 语言规定，如果指针变量 p 已指向数组中的一个元素，则 p+1 指向同一数组中的下一个元素。p+i 和 x+i 等价于 x[i]，或者说它们都指向 x 数组的第 i 个元素。

【地址法】*(p+i)和*(x+i)等价于 x[i]。实际上，编译器会将数组元素 x[i]处理成*(x+i)，即按数组的首地址加上相对位移量得到要找元素的地址，然后找出该单元中的内容。

【指针法】用间接访问的方法来访问数组元素，指向数组的指针变量也可以带下标，如 p[i]与*(p+i)等价。

3）字符串指针和指向字符串的指针变量 在 C 语言中，有两种方法实现一个字符串的运算：一种是使用字符数组；另一种是使用字符串指针。例如：

```
char   a[]={'C', 'o', 'd', 'e', 'V', 'i', 's', 'i', 'o', 'n', 'A', 'V', 'R' , '\0'};
                                                    //使用字符数组定义
char   *b="CodeVisionAVR";                          //使用字符串指针定义
```

字符串指针变量的定义说明与指向字符变量的指针变量说明是相同的。在上述程序段中，a[]是一个字符数组，字符数组是以“\0”常量结尾的；b 是指向字符串的指针变量，它没有定义字符数组。由于 C 语言对字符串常量是按字符数组处理的，所以实际在使用字符串指针时，C 编译器也在内存中开辟了一个字符数组用于存放字符串常量。

📖 注意:

用字符数组和字符串指针变量都可实现字符串的存储和运算，但二者是有区别的。在使用时应注意以下 4 个问题。

☺ 字符串指针变量本身是一个变量，用于存放字符串的首地址。而字符串本身存放在以该首地址为首的一块连续的内存空间中，并以'\0'作为串的结束。字符数组是由若干个数组元素组成的，它可用于存放整个字符串。

☺ 定义一个字符数组时，在编译中已分配内存单元，有确定的地址。而定义一个字符指针变量时，会给指针变量分配内存单元，但并不知道该指针变量具体指向哪个字符串，即指针变量存放的地址不确定。

☺ 赋值方式不同。对字符数组不能整体赋值，只能将其转化成分量后，对单个元素进行赋值。而字符串指针变量可整体进行赋值，直接将其指向字符串首地址即可。

☺ 字符串指针变量的值在程序运行过程中可以改变，而字符数组名是一个常量，不能改变。



任务 5　函数与编译预处理

1．函数

一个较大的程序通常由多个程序模块组成，每个模块用来实现一个特定的功能。在程序设计中，模块的功能是用子程序来实现的。在 C 语言中，子程序的作用是由函数来完成的。因此，C 语言是由函数构成的，函数是 C 语言中的一种基本模块。一个 C 语言程序由一个主函数和若干个函数构成，由主函数调用其他函数，其他函数也可以互相调用。同一个函数可以被一个或多个函数调用任意次，同一工程中的函数也可以分放在不同文件中一起编译。

从使用者的角度来看，函数分为两种，即标准库函数和用户自定义函数。标准库函数是由 C 编译系统的函数库提供的，用户不需自己定义这些函数，可以直接使用它们。用户自定义函数是由用户根据自己的需要编写的函数，用于解决用户的专门需要。

从函数的形式看，有 3 种函数，即无参函数、有参函数和空函数。无参函数被调用

时，主调函数并不将数据传送给被调用函数，一般用于执行指定的一组操作。无参函数可以带回或不带回函数值，但一般以不带回函数值的居多。有参函数被调用时，在主调函数与被调用函数之间有参数传递，即主调函数可以将数据传给被调用函数使用，被调用函数中的数据也可以带回来供主调函数使用。空函数的函数体内无语句，为空白的。调用空函数时，什么工作都不做，不起任何作用。定义空函数的目的并不是为了执行某种操作，而是为了以后程序功能的扩充做准备。

1）函数定义的一般形式

【无参函数的定义形式】无参函数的定义形式如下所述。

```
返回值类型标志符　函数名（）
  {
     函数体语句
  }
```

其中，返回值类型标志符指明本函数返回值的类型；函数名是由用户定义的标志符；“()”内没有参数，但该括号不能少，或者括号里加“void”关键字；“{}”中的内容称为函数体语句。在很多情况下，无参函数没有返回值，因此函数返回值类型标志符可以省略。此时，函数返回值类型标志符也可以写为“void”。例如：

```
void   Timer0_I int(void)       //Timer0 初始化函数
  {
TCCR0 = 0x0D;
       TCNT0 = 0x00;
       OCR0   = 78;
       TIMSK |=0x02;
       TIFR   |=0x02 ;
}
```

【有参函数的定义形式】有参函数的定义形式如下所述。

```
返回值类型标志符　函数名（形式参数列表）
形式参数说明
  {
     函数体语句
  }
```

有参函数比无参函数多了一个内容，即形式参数列表。在形式参数列表中给出的参数称为形式参数（简称形参），它们可以是各种类型的变量，各参数之间用逗号间隔。在进行函数调用时，主调函数将赋予这些形式参数实际的值。例如：

```
int min(int j，k)
{
int   n;
       if (j>k)
            {
              n=k;
            }
```

```
        else
            {
n=j;
}
        return   n;
}
```

在此定义了一个 min 函数，返回值为一个整型（int）变量，形式参数为 j 和 k，也都是整型变量。int n 语句定义 n 为一个整型变量，通过 if 条件语句，将最小的值传送给 n 变量。return n 的作用是将 n 的值作为函数值带回到主调函数中，即 n 的返回值。

【空函数的定义形式】空函数的定义形式如下所述。

```
返回值类型标志符   函数名（）
  {   }
```

调用该函数时，实际上什么工作都不用做，它没有任何实际用途。例如:

```
float   min()
  {   }
```

2）函数的返回值 在函数调用时，通过主调函数的实参与被调函数的形参之间进行数据传递来实现函数间的参数传递。在被调函数的最后，通过 return 语句返回，函数将被调函数中的确定值返回给主调函数。return 语句的一般形式如下：

```
return   (表达式);
```

例如：

```
int x，y;                    //定义两个整型变量 x，y
{
    return(x<y?   x:y);    //如果 x 小于 y，则返回 x，否则返回 y
}
```

函数返回值的类型一般在定义函数时用返回类型标志符来指定。在 C 语言中，凡不加类型说明的函数，都按整型来处理。如果函数值类型的 return 语句中表达式的值不一致，则以函数类型为准，自动进行类型转换。

对于不需要有返回值的函数，可以将该函数定义为“void”类型（或称“空类型”）。这样，编译器会保证在函数调用结束时不使用函数返回任何值。为了使程序减少出错，保证函数的正确调用，凡是不要求有返回值的函数，都应该将其定义为 void 类型。例如：

```
void   abc();                //函数 abc( )为不带返回值的函数
```

3）函数的调用 在 C 语言程序中，函数可以相互调用。所谓函数调用就是指在一个函数体中引用另外一个已经定义了的函数，前者称为主调函数，后者称为被调函数。

在 C 语言中，主调函数通过函数调用语句来使用函数。函数调用的一般形式如下：

```
函数名   (实参列表);
```

对于有参数型的函数，如果包含了多个实参，则应将各参数之间用逗号分隔开。主调用函数的实参的数目与被调用函数的形参的数目应该相等，且类型保持一致。实参与形参按

顺序对应，一一传递数据。

如果调用的是无参函数，则实参表可以省略，但是函数名后面必须有一对空括号。

在 C 语言中，主调用函数调用被调函数可以采用以下 3 种调用方式之一。

【函数语句调用】在主调用函数中将函数调用作为一条语句，并不要求被调用函数返回结果数值，只要求函数完成某种操作，例如:

```
disp_LED( );          //无参调用，不要求被调函数返回一个确定的值，只要求此函数完成
                      //LED 显示操作
```

【函数表达式调用】函数作为表达式的一项出现在表达式中，要求被调用函数带有 return 语句，以便返回一个明确的数值参加表达式的运算。例如:

```
a = 3*min(x，y);          //被调用函数 min 作为表达式的一部分，它的返回值乘 3 再赋给 a
```

【作为函数参数调用】在主调函数中将函数调用作为另一个函数调用的实参。例如:

```
a = min(b，min(c，d))     //min(c，d)是一次函数调用，它的值作为另一次调用的实参。a 为 b、
                          //c 和 d 中的最小值
```

在一个函数中调用另一函数（即被调用函数）时，需具备以下条件。

☺ 被调用函数必须是已经存在的函数（是库函数或用户自己定义的函数）。

☺ 如果程序中使用了库函数，或者使用了不在同一文件中的其他自定义函数，则应该在程序的开头处使用#include 包含语句，以将所有的函数值包括到程序中来。

☺ 对于自定义函数，如果该函数与调用它的函数在同一文件中，则应根据主调用函数与被调用函数在文件中的位置，决定是否对被调用函数的类型做出说明。这种类型说明的一般形式为

```
返回值类型说明符　被调用函数的函数名( );
```

在 C 语言中，在以下 3 种情况下可以不在调用函数前对被调用函数做类型说明。

☺ 如果函数的值（函数的返回值）为整型或字符型，可以不必进行说明，系统会对它们自动按整型进行说明。

☺ 如果被调用函数的定义出现在主调用函数前，可以不对被调用函数加以说明。因为 C 编译器在编译主调用函数前，已经预先知道已定义的被调用函数的类型，并会自动对其加以处理。

☺ 如果已在所有函数定义前，在文件的开头、函数的外部说明了函数类型，则在各个主调函数中不必对所调用的函数再做类型说明。

4）函数的嵌套调用与递归调用

【函数的嵌套调用】在 C 语言中，函数的定义都是相互独立的，不允许在定义函数时，一个函数内部包含另一个函数。虽然在 C 语言中函数不能嵌套定义，但可以嵌套调用函数。嵌套调用函数是指在一个函数内调用另一个函数，即在被调用函数中又调用其他函数。

在 CodeVisionAVR 编译器中，函数间的调用及数据保存与恢复是通过硬件堆栈和软件堆栈来实现的。当没有使用外部数据存储器时，硬件堆栈和软件堆栈均在内部数据存储器中；当有外部数据存储器时，则硬件堆栈在内部数据存储器中，软件堆栈在外部数据存储器中。在 CodeVisionAVR 编译器中，嵌套层数只受到硬件堆栈和软件堆栈的限制，如果嵌套

层数太深，有可能导致硬件或软件堆栈溢出。

【函数的递归调用】在调用一个函数的过程中又出现直接或间接调用该函数本身，称为函数的递归调用。在 C 语言中，允许函数递归调用。函数的递归调用通常用于问题的求解，可以将一种解法逐次地用于问题的子集表示的场合。CodeVisionAVR 编译器能够自动处理函数递归调用的问题。在递归调用时不必做任何声明，其调用深度仅受到堆栈大小的限制。

2．编译预处理

编译预处理是 C 语言编译系统的一个重要组成部分。很好地利用 C 语言的预处理命令可以增强代码的可读性、灵活性，使其易于修改，并便于程序的结构化。在 C 语言程序中，凡是以“#”开头的均表示这是一条预处理命令语句，如包含命令#include、宏定义命令#define 等。C 提供的预处理功能有 3 种，即宏定义、文件包含和条件编译。

1）宏定义 宏定义命令为#define，它的作用是实现用一个简单易读的字符串来代替另一个字符串。宏定义可以增强程序的可读性和维护性。宏定义分为不带参数的宏定义和带参数的宏定义。

【不带参数的宏定义】不带参数的宏定义，其宏名后不带参数。不带参数宏定义的一般形式为

```
#define  标志符  字符串
```

其中，“#”表示这是一条预处理命令；“define”表示为宏定义命令；“标志符”为所定义的宏名；“字符串”可以是常数、表达式等。例如：

```
#define  PI  3.1415926
```

该语句的作用是指定用标志符（即宏名）“PI”代替“3.1415926”字符串，这种方法使用户能以一个简单的标志符代替一个长的字符串。当程序中出现 3.1415926 这个常数时，就可以用“PI”这个字符串来代替了。如果想修改这个常数，只需要修改这个宏定义中的常数即可，这就是增加程序的维护性的体现。

📖 说明：

☺ 宏定义是用宏名代替一个字符串，在宏展开时又以该字符串取代宏名，因此它是一种简单的替换。通过这种宏定义的方法，可以减少程序中重复书写某些字符串的工作量。字符串中可以包含任何字符、常数或表达式，预处理程序对它不进行任何检查。

☺ 宏名可以用大写或小写字母表示，但为了区别于一般的变量名，通常采用大写字母。

☺ 宏定义不是 C 语句，不用加分号；如果加分号，则在编译时会连同分号一起转换。

☺ 当宏定义在一行中书写不下，需要在下一行继续书写时，应该在最后一字符后紧跟着加一个反斜线“\”，并在新的一行的起始位置继续书写，起始位置不能插入空格。

☺ 可以用#undef 终止一个宏定义的作用域。

☺ 一个宏命令只能定义一个宏名。

【带参数的宏定义】带参数的宏在预编译时不仅要进行字符串替换，还要进行参数替换。带参数的宏定义的一般形式为

```
#define  宏名（形参表）字符串
```

带参数的宏调用的一般形式：

```
宏名 (实参表);
```

例如：

```
#define MIN(x，y)  ((x)<(y)) ? (x):(y))      //宏定义
a=MIN(3，7)                                  //宏调用
```

📖 说明：

☺ 在带参数的宏定义中，宏名和形参表之间不能有空格出现，否则会将空格后的字符都当做替换字符串的一部分。

☺ 在宏定义中，字符串内的形参最好用“()”括起来，以避免出错。

☺ 带参数的宏与函数是不同的。

- 函数调用时，会先求出表达式的值，然后代入形参；而使用带参数的宏只是进行简单的字符替换，在宏展开时并不求解表达式的值。
- 函数调用是在程序运行时处理的，会分配临时的内存单元；而使用带参数的宏只是在编译时进行处理，在展开时并不分配内存单元，不进行值的传递处理，也没有“返回值”的概念。
- 对函数中的实参和形参都要定义类型，二者的类型要求一致，若不一致，应进行类型转换；而宏不存在类型问题，宏名无类型，它的参数也无类型，只是一个符号而已，展开时代入指定的字符即可。
- 调用函数只能得到一个返回值，而使用宏可以设法得到多个结果。

2）文件包含　所谓“文件包含”是指一个源文件可以将另外一个源文件的全部内容包含进来，即将另外的文件包含到本文件中。C 语言中的“#include”为文件包含命令，其一般形式为

```
#include <文件名>
```

或

```
#include “文件名”
```

例如：

```
#include <mega16.h>
#include <stdio.h>
#include <delay.h>
```

上述程序的文件包含命令的功能是将“mega16.h”文件插入#include <mega16.h>位置；将“stdio.h”文件插入也就是说#include <stdio.h>命令行位置；将“delay.h”文件插入#include<delay.h>命令行位置。也就是说在编译预处理时，源程序将“mega16.h”、“stdio.h”和“delay.h”这 3 个文件的全部内容复制并分别插入#include <mega16.h>、#include <stdio.h>、#include <delay.h>命令行位置。

在程序设计中，文件包含是很有用的。它可以节省程序设计人员的重复工作，或者可以先将一个大的程序分成多个源文件，由多个编程人员分别编写程序，然后再用文件包含命

令把源文件包含到主文件中。

📖 注意:

☺ 在#include 命令中，文件名可以用双引号或尖括号的形式括起来，但这两种形式有所区别：采用双引号将文件括起来时，系统首先在引用被包含文件的源文件所在的 C 文件目录中寻找要包含的文件，如果找不到，再按系统指定的标准方式搜索\inc 目录；使用尖括号将文件括起来时，不检查源文件所在的文件目录而直接按系统指定的标准方式搜索\inc 目录。

☺ 一个#include 命令只能指定一个被包含文件，如果要包含多个文件，则需要使用多个 include 命令。

☺ #include 命令行包含的文件称为“头文件”。头文件名可以由用户指定，也可以是系统头文件，其后缀名为.h。

☺ 在一个被包含的文件中同时又可以包含另一个被包含的文件，即文件包含可以嵌套。通常，嵌套有深度的限制，这种限制根据编译器的不同而不同。在 CodeVisionAVR C 编译器中，最多允许 16 层文件的嵌套。

☺ 当被包含文件修改后，对包含该文件的源程序必须重新进行编译。

☺ #include 语句可以位于代码的任何位置，但它通常设置在程序模块的顶部，以提高程序的可读性。

3）条件编译 通常情况下，在编译器中单击文件编译时，将会对源程序中所有的行都进行编译（注释行除外）。如果程序员想要源程序中的部分内容只在满足一定条件时才进行编译，可通过“条件编译”对一部分内容指定编译的条件来实现相应操作。条件编译命令有以下 3 种形式。

【第 1 种形式】

```
#ifdef  标志符
   程序段 1
#else
   程序段 2
#endif
```

其作用是，当标志符已经被定义过（通常是用#define 命令定义）时，只对程序段 1 进行编译，否则编译程序段 2。如果没有程序段 2，则本格式中的“#else”可以没有。程序段 1 可以是语句组，也可以是命令行。

【第 2 种形式】

```
#ifndef  标志符
   程序段 1
#else
   程序段 2
#endif
```

其作用是，当标志符没有被定义时，只对程序段 1 进行编译，否则编译程序段 2。这种形式的作用与第 1 种形式的作用正好相反。在书写上将第 1 种形式中的“#ifdef”改为“#ifndef”即可。

【第 3 种形式】

```
#if  常量表达式
  程序段 1
#else
  程序段 2
#endif
```

其作用是，如果常量表达式的值为逻辑“真”，则对程序段 1 进行编译，否则编译程序段 2。可以事先给定一定条件，使程序在不同的条件下执行不同的功能。

项目四　LED 广告灯的设计

〖知识目标〗
☺ 单片机 I/O 端口及端口的基本应用;
☺ CVAVR C 的基本结构及设计方法;
☺ 程序对单片机输出端口的控制方法。
〖能力目标〗
☺ 能根据设计任务要求编制程序流程图，理解程序对发光二极管（LED）的控制原理;
☺ 能绘制 LED 广告灯电路原理图;
☺ 会用 CVAVR 软件对源程序进行编译调试，并与 Proteus 软件联调，实现电路仿真。

广告灯是一种常见的装饰，常用于街上的广告及舞台装饰等场合。常见的 LED 广告灯主要包括闪烁灯、流水广告灯、拉幕式与闭幕式广告灯，以及复杂广告灯等。

任务 1　LED 控制原理认知

1．LED 简介

发光二极管（Light Emitting Diode，LED）是一种由磷化镓（GaP）等半导体材料制成的、能直接将电能转变成光能的发光显示器件。当其内部有一定电流通过时，它就会发光。LED 的心脏是一个半导体的 LED 芯片，芯片的一端附在一个支架上，一端是负极，另一端连接电源的正极，使整个晶片被圆形环氧树脂封装起来，其构成如图 4-1 所示。

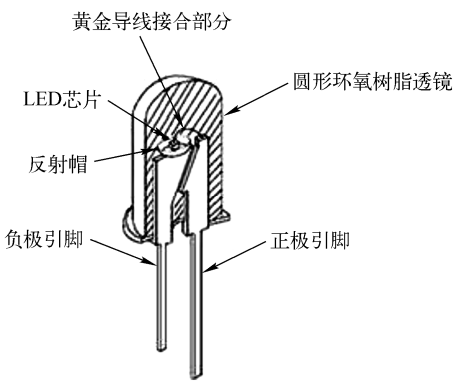


图 4-1　LED 的构成

半导体芯片由两部分组成，一部分是 P 型半导体，在它里面空穴占主导地位，另一端是 N 型半导体，在这边主要是电子。但这两种半导体连接起来时，它们之间就形成一个 PN 结。当电流通过导线作用于这个晶片时，电子就会被推向 P 区，在 P 区里电子跟空穴复合，然后就会以光子的形式发出能量，这就是 LED 的发光原理。而光的波长也就是光的颜色，是由形成 PN 结的材料决定的。

LED 还可分为普通单色 LED、高亮度 LED、超高亮度 LED、变色 LED、闪烁 LED、电压控制型 LED、红外 LED 和负阻等。本项目使用的是普通单色 LED，所以，在此只讲解普通单色 LED 的相关知识。

普通单色 LED 主要有发红光、绿光、蓝光、黄光的 LED。它们具有体积小、工作电压低、工作电流小、发光均匀稳定、响应速度快、寿命长等优点，可用各种直流、交流、脉冲等电源驱动点亮。它们属于电流控制型半导体器件，使用时需串接合适的限流电阻。

普通单色 LED 的发光颜色与发光的波长有关，而发光的波长又取决于制造 LED 所用的半导体材料。红色 LED 的波长一般为 650～700nm，琥珀色 LED 的波长一般为 630～650nm，橙色 LED 的波长一般为 610～630nm，黄色 LED 的波长一般约为 585nm，绿色 LED 的波长一般为 555～570nm。

常用的国产普通单色 LED 有 BT（厂标型号）系列、FG（部标型号）系列和 2EF 系列；常用的进口普通单色 LED 有 SLR 系列和 SLC 系列等。

2．ATmega16 单片机 I/O 端口控制 LED 的方法

1）ATmega16 单片机 I/O 端口　ATmega16 芯片有 PORTA、PORTB、PORTC、PORTD（简称 PA、PB、PC、PD）4 组 8 位共 32 路通用 I/O 端口，分别对应于芯片上的 32 个 I/O 引脚。所有这些 I/O 端口都具有双功能复用，有的还具有 3 功能复用。其中第 1 功能均作为通用数字 I/O 接口使用，而复用功能则分别用于中断、定时器/计数器、USRAT、I^2C 通信、SPI 通信、模拟比较、捕捉等应用场合。

作为通用数字 I/O 使用时，所有的 I/O 端口都具有真正的读-修改-写功能。这意味着用 SBI 或 CBI 指令改变某些引脚的方向（或者是端口电平、禁止/使能上拉电阻）时不会无意地改变其他引脚的方向（或者是端口电平、禁止/使能上拉电阻）。

【PA 端口】PA 端口为一个 8 位的双向 I/O 端口，它有 3 个 I/O 寄存器地址，分别为数据寄存器 PORTA、数据方向寄存器 DDRA 和 PA 端口的输入引脚寄存器 PINA。其中，PORTA 的 I/O（数据）地址为 0x1B（0x003B）；DDRA 的 I/O（数据）地址为 0x1A（0x003A）；PINA 的 I/O（数据）地址为 0x19（0x0039）。PINA 为只读寄存器，而 DDRA 和 PORTA 为可读/写寄存器。对 PINA 寄存器的某一位写逻辑“1”将造成数字寄存器的相应位发生“0”与“1”的翻转。

当 DDRA=1 时，I/O 端口处于输出工作方式，此时数据寄存器 PORTA 中的数据通过一个推挽电路输出到外部引脚。AVR 的输出采用推挽电路是为了提高 I/O 端口的输出能力。当 PORTA=1 时，I/O 端口呈现高电平，同时可以吸收 20mA 的电流以直接驱动 LED 显示。

当 DDRA=0 时，I/O 端口处于输入工作方式，输入引脚寄存器 PINA 中的数据就是外部引脚的实际电平，通过读 I/O 指令可将物理引脚的真实数据读入 MCU。此外，当 I/O 端口定义为输入时（DDRA=0），通过 PORTA 的控制，可设置是否使用内部的上拉电阻。

当特殊功能寄存器 SFIOR 的上拉禁止位 PUD 被置位时，所有端口的引脚都被禁止。不论如何配置 DDRA*n*，都可以通过读引脚 PINA*n* 来获得该引脚上的逻辑电平。PA 端口的引脚配置见表 4-1。

表 4-1　PA 端口的引脚配置

DDRA*n*	PORTA*n*	PUD	I/O 方式	是否使用内部上拉电阻	引脚状态说明
0	0	×	输入	否	高阻态（Hi-Z）
0	1	0	输入	是	被外部电路拉低时的输出电流（μA）
0	1	1	输入	否	高阻态（Hi-Z）

续表

DDRA*n*	PORTA*n*	PUD	I/O 方式	是否使用内部上拉电阻	引脚状态说明
1	0	×	输出	否	推挽低电平输出，吸收电流（20mA）
1	1	×	输出	否	推挽高电平输出，输出电流（20mA）

PA 端口引脚具有与可选的外部数据 SRAM 有关的第 2 功能，即 PA 端口在访问外部数据存储器时，可以配置为复用的低位地址/数据线。在该模式下，PA 端口有内部的上拉电阻。与 PA 端口有关的寄存器及输出引脚见表 4-2。

表 4-2　与 PA 端口有关的寄存器及输出引脚

寄存器	寄存器的相关位及状态								
PORTA	bit	7	6	5	4	3	2	1	0
	位名称	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0
	读/写状态	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
	初始值	0	0	0	0	0	0	0	0
DDRA	bit	7	6	5	4	3	2	1	0
	位名称	DDRA7	DDRA 6	DDRA 5	DDRA 4	DDRA 3	DDRA 2	DDRA 1	DDRA 0
	读/写状态	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
	初始值	0	0	0	0	0	0	0	0
PINA	bit	7	6	5	4	3	2	1	0
	位名称	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0
	读/写状态	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
	初始值	Hi-Z	Hi-Z	Hi-Z	Hi-Z	Hi-Z	Hi-Z	Hi-Z	Hi-Z

【PB 端口】PB 端口为一个 8 位的双向 I/O 端口，它有 3 个 I/O 寄存器地址，分别为数据寄存器 PORTB、数据方向寄存器 DDRB 和 PB 端口的输入引脚寄存器 PINB。其中，PORTB 的 I/O（数据）地址为 0x18（0x0038）；DDRB 的 I/O（数据）地址为 0x17（0x0037）；PINB 的 I/O（数据）地址为 0x16（0x0036）。PINB 为只读寄存器，而 DDRB 和 PORTB 为可读/写寄存器。对 PINB 寄存器的某一位写逻辑“1”将造成数字寄存器的相应位发生“0”与“1”的翻转。

当 DDRB=1 时，I/O 端口处于输出工作方式，此时数据寄存器 PORTB 中的数据通过一个推挽电路输出到外部引脚。当 PORTB=1 时，I/O 引脚呈现高电平，同时可以吸收 20mA 的电流以直接驱动 LED 显示。

当 DDRB=0 时，I/O 端口处于输入工作方式，输入引脚寄存器 PINB 中的数据就是外部引脚的实际电平，通过读 I/O 指令可将物理引脚的真实数据读入 MCU。此外，当 I/O 端口定义为输入时（DDRB=0），通过 PORTB 的控制，可设置是否使用内部的上拉电阻。

当特殊功能寄存器 SFIOR 的上拉禁止位 PUD 被置位时，所有端口的引脚都被禁止。不论如何配置 DDRB*n*，都可以通过读引脚 PINB*n* 来获得该引脚上的逻辑电平。PB 端口的引脚配置见表 4-3。

表 4-3　PB 端口的引脚配置

DDRB*n*	PORTB*n*	PUD	I/O 方式	是否使用内部上拉电阻	引脚状态说明
0	0	×	输入	否	高阻态（Hi-Z）
0	1	0	输入	是	被外部电路拉低时的输出电流（μA）
0	1	1	输入	否	高阻态（Hi-Z）
1	0	×	输出	否	推挽低电平输出，吸收电流（20mA）
1	1	×	输出	否	推挽高电平输出，输出电流（20mA）

PB 端口引脚具有与可选的外部数据 SRAM 有关的第 2 功能，即 PB 端口在访问外部数据存储器时，可以配置为复用的低位地址/数据线。在该模式下，PB 端口有内部的上拉电阻。与 PB 端口有关的寄存器及输出引脚见表 4-4。

表 4-4　与 PB 端口有关的寄存器及输出引脚

寄存器	寄存器的相关位及状态								
PORTB	bit	7	6	5	4	3	2	1	0
	位名称	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
	读/写状态	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
	初始值	0	0	0	0	0	0	0	0
DDRB	bit	7	6	5	4	3	2	1	0
	位名称	DDRB7	DDRB 6	DDRB 5	DDRB 4	DDRB 3	DDRB 2	DDRB 1	DDRB 0
	读/写状态	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
	初始值	0	0	0	0	0	0	0	0
PINB	bit	7	6	5	4	3	2	1	0
	位名称	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
	读/写状态	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
	初始值	Hi-Z	Hi-Z	Hi-Z	Hi-Z	Hi-Z	Hi-Z	Hi-Z	Hi-Z

【PC 端口】PC 端口为一个 8 位的双向 I/O 端口，它有 3 个 I/O 寄存器地址，分别为数据寄存器 PORTC、数据方向寄存器 DDRC 和 PC 端口的输入引脚寄存器 PINC。其中，PORTC 的 I/O（数据）地址为 0x15（0x0035）；DDRC 的 I/O（数据）地址为 0x14（0x0034）；PINC 的 I/O（数据）地址为 0x13（0x0033）。PINC 为只读寄存器，而 DDRC 和 PORTC 为可读/写寄存器。对 PINC 寄存器的某一位写逻辑“1”将造成数字寄存器的相应位发生“0”与“1”的翻转。

当 DDRC=1 时，I/O 端口处于输出工作方式，此时数据寄存器 PORTC 中的数据通过一个推挽电路输出到外部引脚。当 PORTC=1 时，I/O 引脚呈现高电平，同时可以吸收 20mA 的电流以直接驱动 LED 显示。

当 DDRC=0 时，I/O 端口处于输入工作方式，输入引脚寄存器 PINC 中的数据就是外部引脚的实际电平，通过读 I/O 指令可将物理引脚的真实数据读入 MCU。此外，当 I/O 端口定义为输入时（DDRC=0），通过 PORTC 的控制，可设置是否使用内部的上拉电阻。

当特殊功能寄存器 SFIOR 的上拉禁止位 PUD 被置位时，所有端口的引脚都被禁止。不论如何配置 DDRC*n*，都可以通过读引脚 PINC*n* 来获得该引脚上的逻辑电平。PC 端口的引脚配置见表 4-5。

表 4-5　PC 端口的引脚配置

DDRC*n*	PORTC*n*	PUD	I/O 方式	是否使用内部上拉电阻	引脚状态说明
0	0	×	输入	否	高阻态（Hi-Z）
0	1	0	输入	是	被外部电路拉低时的输出电流（μA）
0	1	1	输入	否	高阻态（Hi-Z）
1	0	×	输出	否	推挽低电平输出，吸收电流（20mA）
1	1	×	输出	否	推挽高电平输出，输出电流（20mA）

PC 端口引脚具有与可选的外部数据 SRAM 有关的第 2 功能，即 PC 端口在访问外部数据存储器时，可以配置为复用的低位地址/数据线。在该模式下，PC 端口有内部的上拉电阻。与 PC 端口有关的寄存器及输出引脚见表 4-6。

表 4-6　与 PC 端口有关的寄存器及输出引脚

寄存器	寄存器的相关位及状态								
PORTC	bit	7	6	5	4	3	2	1	0
	位名称	PORTC7	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0
	读/写状态	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
	初始值	0	0	0	0	0	0	0	0
DDRC	bit	7	6	5	4	3	2	1	0
	位名称	DDRC7	DDRC 6	DDRC 5	DDRC 4	DDRC 3	DDRC 2	DDRC 1	DDRC 0
	读/写状态	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
	初始值	0	0	0	0	0	0	0	0
PINC	bit	7	6	5	4	3	2	1	0
	位名称	PINC7	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0
	读/写状态	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
	初始值	Hi-Z	Hi-Z	Hi-Z	Hi-Z	Hi-Z	Hi-Z	Hi-Z	Hi-Z

【PD 端口】PD 端口为一个 8 位的双向 I/O 端口，它有 3 个 I/O 寄存器地址，分别为数据寄存器 PORTD、数据方向寄存器 DDRD 和 PD 端口的输入引脚寄存器 PIND。其中 PORTD 的 I/O（数据）地址为 0x12（0x0032）；DDRD 的 I/O（数据）地址为 0x11（0x0031）；PIND 的 I/O（数据）地址为 0x10（0x0030）。PIND 为只读寄存器，而 DDRD 和 PORTD 为可读/写寄存器。对 PIND 寄存器的某一位写逻辑“1”将造成数字寄存器的相应位发生“0”与“1”的翻转。

当 DDRD=1 时，I/O 端口处于输出工作方式，此时数据寄存器 PORTD 中的数据通过一个推挽电路输出到外部引脚。当 PORTD=1 时，I/O 引脚呈现高电平，同时可以吸收 20mA 的电流以直接驱动 LED 显示。

当 DDRD=0 时，I/O 端口处于输入工作方式，输入引脚寄存器 PIND 中的数据就是外部引脚的实际电平，通过读 I/O 指令可将物理引脚的真实数据读入 MCU。此外，当 I/O 端口定义为输入时（DDRD=0），通过 PORTD 的控制，可设置是否使用内部的上拉电阻。

当特殊功能寄存器 SFIOR 的上拉禁止位 PUD 被置位时，所有端口的引脚都被禁止。不论如何配置 DDRD*n*，都可以通过读引脚 PIND*n* 来获得该引脚上的逻辑电平。PD 端口的引脚配置见表 4-7。

表 4-7　PD 端口的引脚配置

DDRD*n*	PORTD*n*	PUD	I/O 方式	是否使用内部上拉电阻	引脚状态说明
0	0	×	输入	否	高阻态（Hi-Z）
0	1	0	输入	是	被外部电路拉低时的输出电流（μA）
0	1	1	输入	否	高阻态（Hi-Z）
1	0	×	输出	否	推挽低电平输出，吸收电流（20mA）
1	1	×	输出	否	推挽高电平输出，输出电流（20mA）

PD 端口引脚具有与可选的外部数据 SRAM 有关的第 2 功能，PD 端口在访问外部数据存储器时，可以配置为复用的低位地址/数据线。在该模式下，PD 端口有内部的上拉电阻。与 PD 端口有关的寄存器及输出引脚见表 4-8。

表 4-8　与 PD 端口有关的寄存器及输出引脚

寄存器	寄存器的相关位及状态								
PORTD	bit	7	6	5	4	3	2	1	0
	位名称	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
	读/写状态	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
	初始值	0	0	0	0	0	0	0	0
DDRD	bit	7	6	5	4	3	2	1	0
	位名称	DDRD7	DDRD 6	DDRD 5	DDRD 4	DDRD 3	DDRD 2	DDRD 1	DDRD 0
	读/写状态	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
	初始值	0	0	0	0	0	0	0	0
PIND	bit	7	6	5	4	3	2	1	0
	位名称	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
	读/写状态	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
	初始值	Hi-Z	Hi-Z	Hi-Z	Hi-Z	Hi-Z	Hi-Z	Hi-Z	Hi-Z

2）限流电阻的选取　PA～PD 端口的第一个功能都是双向数字 I/O 端口，其中每一位都可以由指令设置为独立的输入口或输出口中。当 I/O 端口设置为输入方式时，引脚内部还配有上拉电阻，这个内部的上拉电阻可通过编程设置为上拉有效或上拉无效。

如果 AVR 的 I/O 端口设置为输出方式工作，则当输出高电平时，能够输出 20mA 的电流；而当其输出低电平时，可以吸收 40mA 的电流。因此，AVR 单片机的 I/O 端口驱动能力非常强，能够直接驱动 LED、数码管等。

通常 LED 的导通电压降大于 1.5V、导通电流大于 5mA 时，人的眼睛就可以明显地观察到 LED 发光，导通电流越大，亮度越高。如果导通电流大于 10mA，且没有外接限流电阻，LED 在工作时会迅速发热，为了防止 LED 过热损害，必须串联限流电阻以限制 LED 的功耗。表 4-9 所列为典型的 LED 功率指标。

表 4-9　典型的 LED 功率指标

参　数	红色 LED	绿色 LED	黄色 LED	橙色 LED
最大功率/mW	55	75	60	75
最大正向电流/mA	160	100	80	100
最大恒定电流/mA	25	25	20	25

LED 的发光功率可以由其两端的电压和通过 LED 的电流来计算得到，公式如下：$P_d = U_d \times I_d$。其中 U_d 为 LED 的正向电压；I_d 为正向电流。

普通单色 LED 的正向压降一般为 1.5～2.0V，其中，红色 LED 约为 1.6V，绿色 LED 约为 1.7V，黄色 LED 约为 1.8V，蓝色 LED 为 2.5～3.5V 等，它们的反向击穿电压约 5V，正向工作电流一般为 5～20mA。

LED 的典型 U-I 特性曲线如图 4-2 所示。从图中可以看出，LED 的 U-I 特性曲线很陡，使用时，根据 LED 亮度的需要而串联限流电阻 R 以控制通过 LED 的电流大小。为了保护单片机的驱动输出引脚，通过 LED 的正向工作电流一般应限制在约 10mA，正向电压限制在约 2V。

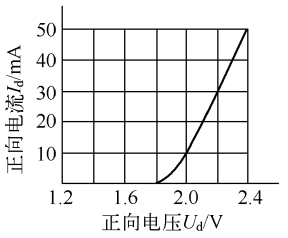


图 4-2　LED 发光二极管伏-安特性曲线

限流电阻 R 可用下式计算：$R = (E - U_d)/I_d$，E 为电源电压，由于单片机使用的电压通常为 5V，因此 E 取 5V。

例如，若限制电流 I_d 为 10mA，LED 的正向电压 U_d 约为 2V，从而得到限流电阻值 $R = (5\text{V} - 2\text{V})/10\text{mA} = 300(\Omega)$。

在实际应用中，为了有效保护单片机驱动输出引脚，应预留一定的安全系数，一般对 LED 驱动采用的限流电阻都要比采用 10mA 计算出的大，常用的典型值为 470Ω。由于 AVR 的 I/O 端口输出“0”时，可以吸收最大 40mA 的电流，因此采用控制 LED 负极的设计比较好。

任务 2　闪灯广告灯的设计

用 ATmega16 的 PC 口作为输出口，外接 LED D1 和 D2。编写程序，实现两个 LED 互闪。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。执行菜单命令“File”→“New Design”，新建一个 DEFAULT 模板，保存文件名为“闪烁灯.DSN”。在元器件选择按钮

P L DEVICES 中单击“P”按钮，或执行菜单命令“Library”→“Pick Device/Symbol”，添加表 4-10 所列的元器件。

表 4-10　闪烁灯所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	电解电容 CAP-ELEC 10μF	电阻 RES
晶振 CRYSTAL 8MHz	发光二极管 LED-BLUE	发光二极管 LED-RED	

在 ISIS 原理图编辑窗口中放置元器件，再单击工具箱中元器件终端图标，在对象选择器中单击 POWER 和 GROUND，放置电源和地。放置好元器件后，布好线。双击各元器件，设置相应的元器件参数，完成电路图的设计，如图 4-3 所示。

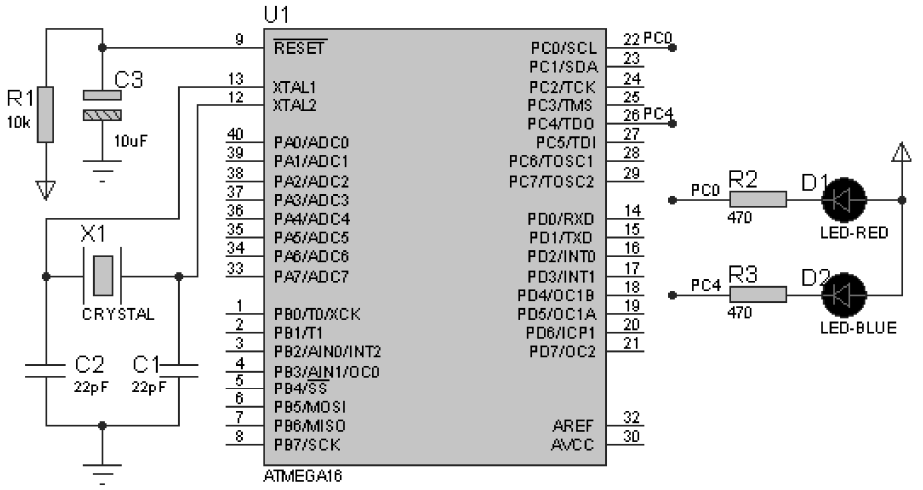


图 4-3　闪烁灯电路图

软件设计

从图 4-3 中可以看出，PC 端口 PA0 和 PA4 端口分别外接 D1 和 D2 的阴极，D1 和 D2 的阳极共接电源，R2 和 R3 分别为 D1 和 D2 的限流电阻。若要 D1 或 D2 点亮，PA0 和 PA4 端口必须处于低电平状态。要实现 D1 和 D2 的互闪控制，可以先将 D1 的状态置为 0（D1 点亮），而 D2 的状态置为 1（D2 熄灭），然后每隔一段时间将它们的状态取反即可。闪烁灯程序流程图如图 4-4 所示。

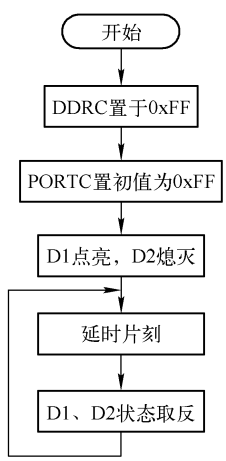


图 4-4　闪烁灯程序流程图

源程序

```
/*****************************************************
  文件名:          闪烁灯.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
```

```
#define uchar unsigned char
#define uint unsigned int
#define   redLED       PORTC.0
#define   blueLED      PORTC.4
void delay(uint k)                              //约 1ms 延时函数
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void main(void)
{
     DDRC=0xFF;
    PORTC=0xFF;
     redLED=0;
    blueLED=1;
     while(1)
      {
         delay(1000);
         redLED=~redLED;
        blueLED=~blueLED;
      }
}
```



调试与仿真

【HEX 或 COFF 文件的生成】

（1）打开 AVR 单片机程序编译软件 CVAVR，执行菜单命令“File”→“NEW”，在弹出的“Create New File”对话框中选中“Project”选项，单击“OK”按钮，弹出“Confirm”对话框，在此对话框中单击“NO”按钮，弹出“Create New Project”对话框。

（2）在“File name”栏中输入新建项目名，单击“Save”按钮，在弹出的“Configure Project”对话框中单击“Cancel”按钮，不采用 CVAVR 项目向导，由用户直接新建项目。

（3）执行菜单命令“File”→“NEW”，在弹出的“Create New File”对话框中选中“Source”选项，再单击“OK”按钮，打开一个空的文本编辑窗口，在此窗口中输入程序，创建新的源程序“PA 端口的应用.C”文件。

（4）执行菜单命令“Project”→“Configure”，弹出“Configure Project”对话框。在此对话框的“Files”选项卡中单击“ADD”按钮，添加刚创建的源程序“PA 端口的应用.C”文件，再单击“Open”按钮。

（5）执行菜单命令“Project”→“Configure”，弹出“Configure Project”对话框。在此对话框的“C Compiler”选项卡中将“Clock”栏设置为 8MHz，将“Chip”栏设置为“ATmega16”。

（6）执行菜单命令“Project”→“Build All”，或者在工具栏中单击图标，进行源程

序的编译。若程序编译成功，将生成“闪烁灯.HEX”和“闪烁灯.COFF”文件。

【具体的调试与仿真】

（1）在 Proteus ISIS 编辑窗口中用鼠标右键将 ATmega16 单片机选中，单击鼠标左键，弹出“Edit Component”对话框，在“Program File”栏中单击图标，选择先前用 CVAVR 生成的“闪烁灯.HEX”文件；在“CKOPT”栏中选择“(0)Programmed”，使外接晶体振荡器的最大频率达 16MHz；在“CKSEL Fuses”栏中选择“(0111)Ext.RC 3.0MHz-8.0MHz”，使 ATmega16 可接外部频率为 3～8MHz 范围的晶体振荡器；在“SUT Fuses”栏中选择“01”，表示可连接的外部晶体为石英振荡器；在“Advanced Properties”栏中将第 2 个框里的内容“(Default)”改为 8MHz 的时钟频率。ATmega16 的属性设置具体如图 4-5 所示。

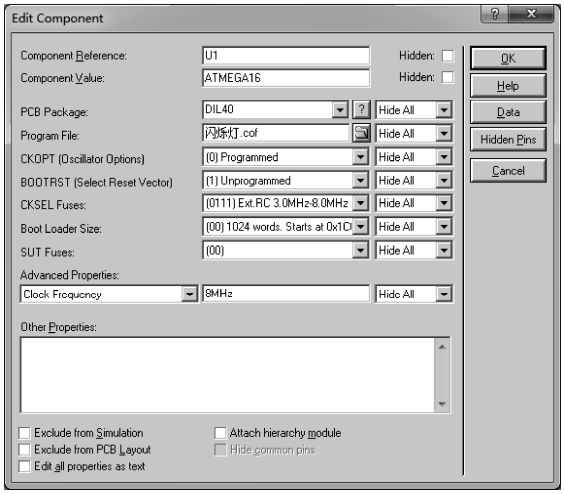


图 4-5　ATmega16 的属性设置

（2）在 Proteus ISIS 中执行菜单命令“File”→“Save Design”，保存设计，生成“PA 端口的应用.DSN”文件。

（3）在 Proteus ISIS 编辑窗口中单击按钮，或者执行菜单命令“Debug”→“Execute”，运行程序。刚运行程序时，D1 亮、D2 熄灭，然后每隔一定的时间 D1 和 D2 的状态取反，其仿真效果如图 4-6 所示。

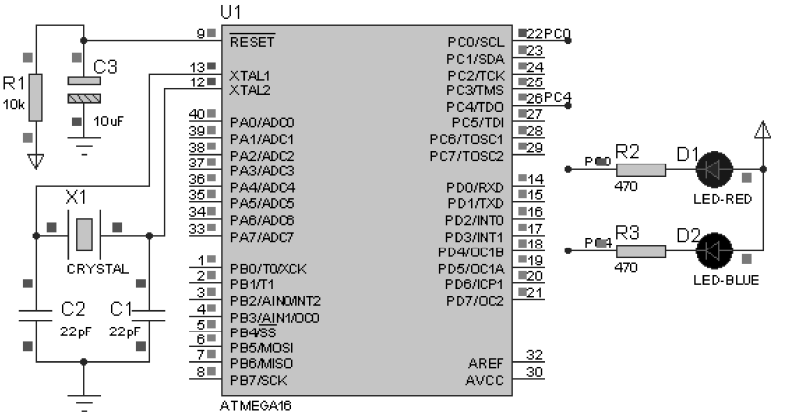


图 4-6　闪烁灯运行仿真效果图

说明：

在 Proteus 中，完成调试与仿真的操作步骤基本相同，因此在本书后续章节的“调试与仿真”部分将不再赘述其详细的操作过程，请参照本节的调试与仿真操作过程进行操作即可。

任务 3　流水广告灯的设计

使用单片机 PC 端口实现 8 只 LED 的流水灯控制。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 4-11 所列的元器件，并完成如图 4-7 所示的硬件电路图设计。

表 4-11　流水灯所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	发光二极管 LED-BIGY	发光二极管 LED-BIBY
电阻 RES	晶振 CRYSTAL 8MHz	发光二极管 LED-BIRG	发光二极管 LED-YELLOW
电解电容 CAP-ELEC			

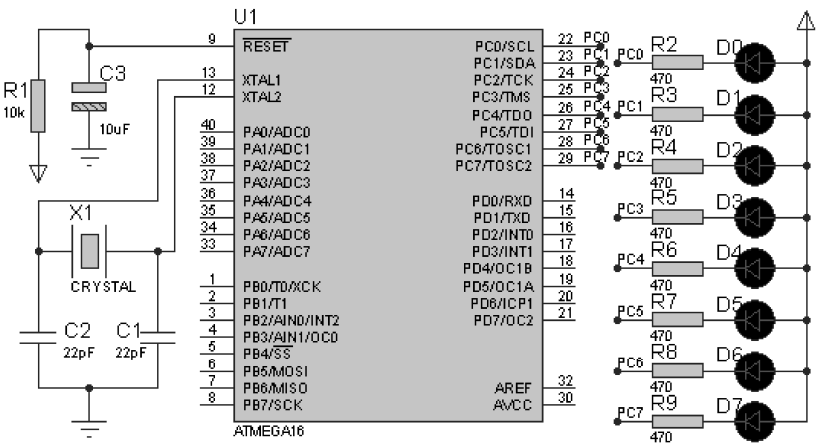


图 4-7　流水灯电路图

软件设计

流水灯又称为跑马灯，是通过单片机控制 8 只 LED D0～D7 循环点亮，即刚开始时 D0 点亮，延时片刻后，接着 D1 亮，然后依次点亮 D2→D3→D4→D5→D6→D7，然后再点亮 D6→D5→D4→D3→D2→D1→D0，重复循环。从显示规律看，流水灯实质上由左移和右移控制实现的，其中 D0→D1→D2→D3→D4→D5→D6→D7 为左移控制；D6→D5→D4→D3→D2→D1→D0 为右移控制。要实现流水灯的左移及右移控制，可以使用移位指令或数组这两种方法进行软件程序设计。

【移位指令实现软件程序设计】在项目三中讲解了按位左移（<<）、按位右移（>>）指令，其中按位左移（<<）用于将一个操作数的各二进制位的全部左移若干位，移位后，空白位补“0”，而溢出的位舍弃；按位右移（>>）用于将一个操作数的各二进制位的全部右移若干位，移位后，空白位补“0”，而溢出的位舍弃。

由于移位指令在移位过程中，空出位自动用“0”填充，而图 4-7 所示的电路中 LED 是低电平有效，因此可以将左移的移位初始数据置为 0x01，右移的移位初始数据置为 0x80，然后将该数据每次移位取反后送给 PORTC，不过在送给 PORTC 前还需判断是否已经移位 8 次，若是，则退出移位操作。其程序流程图如图 4-8 所示。

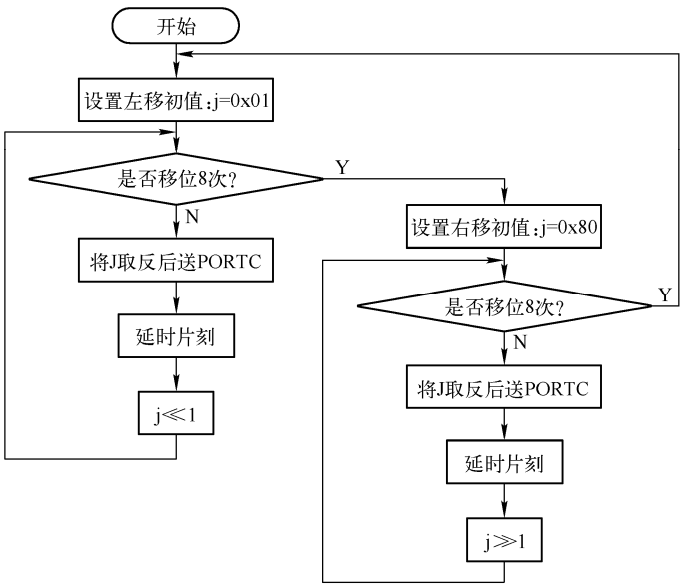


图 4-8　移位指令实现流水灯的程序流程图

【数组实现软件程序设计】使用数组实现流水灯控制，可以将每一时刻的显示状态数据放在一个数组中来实现（见表 4-12），每次通过调用数组中的某一元素内容来控制 LED 的显示状态。

表 4-12　流水灯显示状态数据

LED		D7	D6	D5	D4	D3	D2	D1	D0	PORTC 输出（十六进制）	功能说明
PC 端口		PC7	PC6	PC5	PC4	PC3	PC2	PC1	PC0		
PC 端口输出电平	左移	1	1	1	1	1	1	1	0	0xFE	D0 点亮
		1	1	1	1	1	1	0	1	0xFD	D1 点亮
		1	1	1	1	1	0	1	1	0xFB	D2 点亮
		1	1	1	1	0	1	1	1	0xF7	D3 点亮
		1	1	1	0	1	1	1	1	0xEF	D4 点亮
		1	1	0	1	1	1	1	1	0xDF	D5 点亮
		1	0	1	1	1	1	1	1	0xBF	D6 点亮
		0	1	1	1	1	1	1	1	0x7F	D7 点亮

续表

LED		D7	D6	D5	D4	D3	D2	D1	D0	PORTC 输出（十六进制）	功能说明
PC 端口		PC7	PC6	PC5	PC4	PC3	PC2	PC1	PC0		
PC 端口输出电平	右移	0	1	1	1	1	1	1	1	0x7F	D7 点亮
		1	0	1	1	1	1	1	1	0xBF	D6 点亮
		1	1	0	1	1	1	1	1	0xDF	D5 点亮
		1	1	1	0	1	1	1	1	0xEF	D4 点亮
		1	1	1	1	0	1	1	1	0xF7	D3 点亮
		1	1	1	1	1	0	1	1	0xFB	D2 点亮
		1	1	1	1	1	1	0	1	0xFD	D1 点亮
		1	1	1	1	1	1	1	0	0xFE	D0 点亮

从表 4-13 中可以看出，这些数据使 D0→D7 点亮，就是数据中的二进制数 0 的位置依次左移了 1 位，D7～D0 点亮时，数据中的二进制数 0 的位置依次右移了 1 位。在 CVAVR C 中，要直接实现数据的计算有时不太容易，如果将所有的数据取反后，D0→D7 依次点亮的数据就变成了 0x01，0x02，0x04，0x08，0x10，0x20，0x40，0x80，也就是后一个数是前一个数的基础上乘以 2。所以在实际使用时，通过建立两个一维数组来实现流水灯控制，其程序流程也可以参考图 4-8。

源程序

【移位指令实现的源程序】

```
/*****************************************************
  文件名:          流水灯.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
  说明:            利用移位指令实现
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
void delay(uint k)
{
   uint m,n;
   for(m=0;m<k;m++)
      {
         for(n=0;n<1140;n++)
           {;}
      }
}
void main(void)
{
      uchar i,j;
      DDRC=0xFF;
     PORTC=0xFF;
```

```
    while(1)
     {
       j=0x01;
       for(i=0;i<8;i++)                    //D0～D7 移位点亮（左移）
         {
           PORTC=~j;
           delay(1000);
           j=j<<1;
         }
       j=0x80;
       for(i=0;i<8;i++)                    //D7～D0 移位点亮（右移）
         {
           PORTC=~j;
           delay(1000);
           j=j>>1;
         }
     }
}
```

【数组实现的源程序】

```
/*****************************************************
 文件名:          流水灯.c
 单片机型号:      ATmega16
 晶振频率:        8.0MHz
 说明:            利用数组实现
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
flash uchar discode1[8]={0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80};
flash uchar discode2[8]={0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void main(void)
{
   uchar i,j;
   DDRC=0xFF;
   PORTC=0xFF;
   while(1)
       {
```

```
        for(i=0;i<8;i++)                    //D0～D7 移位点亮（左移）
            {
               j=discode1[i];
               PORTC=~j;
               delay(1000);
            }
        for(i=0;i<8;i++)                    //D7～D0 移位点亮（右移）
            {
               j=discode2[i];
               PORTC=~j;
               delay(1000);
            }
    }
}
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“流水灯.COFF”文件，然后在 Proteus ISIS 中打开已创建的流水灯电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。此时可以看到 D0 亮，延时片刻后，接着 D1 亮，再依次为 D2，D3，D4，D5，D6，D7，然后再点亮 D6，D5，D4，D3，D2，D1，D0，重复循环，其仿真效果如图 4-9 所示。

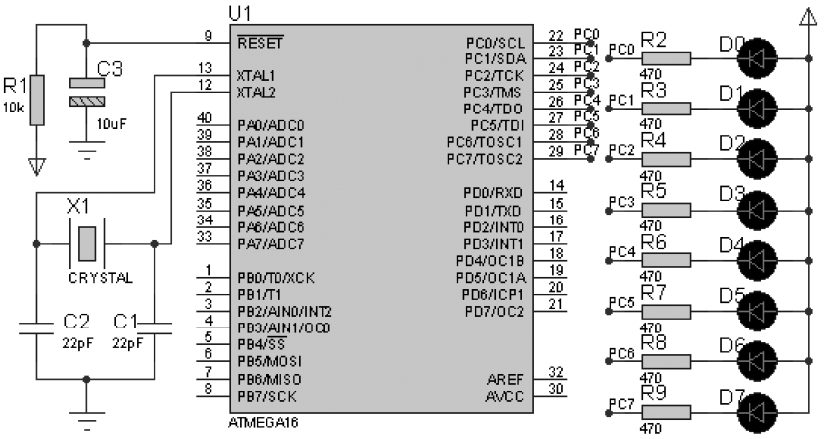


图 4-9　流水灯运行仿真效果图

任务 4　拉幕式与闭幕式广告灯的设计

使用单片机 PC 端口实现 8 只 LED 的拉幕式与闭幕式广告灯设计。

硬件设计

8 只 LED 的拉幕式与闭幕式广告灯设计所使用的硬件电路与流水广告灯的硬件电路完

全相同，因此可参照图 4-7 绘制拉幕式与闭幕式广告灯的电路图。

软件设计

8 只 LED 的拉幕式与闭幕式广告灯设计可分为拉幕式与闭幕式两部分。所谓拉幕式是指 D0～D7 全灭时，延时片刻后首先 D3、D4 亮，其次是 D2、D5 亮，再则是 D1、D6 亮，最后是 D0、D7 亮，从视觉效果上看，好像拉开幕布一样。闭幕式是指 D0～D7 全亮时，延时片刻后首先 D0、D7 灭，其次是 D1、D6 灭，再则是 D2、D5 灭，最后是 D3、D4 灭，其效果就像关闭幕布一样。

从显示规律可以看出，拉幕式时，可分为右移控制和左移控制，即 D3→D2→D1→D0（右移）、D4→D5→D6→D7（左移）；闭幕式时，可分为左移控制和右移控制，即 D0→D1→D2→D3（左移）、D7→D6→D5→D4（右移）。因此，要实现拉幕式与闭幕式广告灯的控制，同样可以使用移位指令、移位函数及数组这 3 种方法进行软件程序设计。

【移位指令实现软件程序设计】使用按位“左移”（<<）、按位“右移”（>>）指令实现拉幕式与闭幕式广告灯控制时，由于硬件电路中 LED 是低电平点亮，所以同样要注意初始数据的设置。在拉幕式控制中，右移初值设为 0x10，左移初值设为 0x08，然后将左移数值与右移数值整合在一起并取反后送给 PC 端口，不过在送给 PC 端口前判断移位次数不再是 8，而应该为 4；闭幕式控制中，左移初值可设为 0x01，右移初值可设为 0x80，然后将左移数值与右移数值整合在一起不需取反而送给 PC 端口，同时左移数值应该加上 0x01，右移数值应该加上 0x80。其程序流程图如图 4-10 所示。

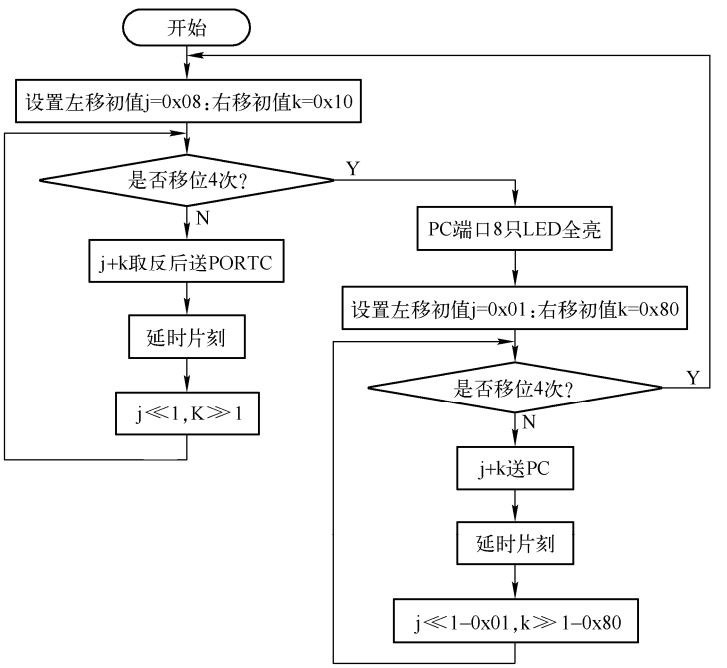


图 4-10　移位指令实现拉幕式与闭幕式广告灯的程序流程图

【数组实现软件程序设计】使用数组实现拉幕式与闭幕式广告灯控制时，数组中的显示状态数据见表 4-13。

表 4-13　拉幕式与闭幕式显示状态数据

LED		D7	D6	D5	D4	D3	D2	D1	D0	PORTC 输出（十六进制）	功能说明
PC 端口		PC7	PC6	PC5	PC4	PC3	PC2	PC1	PC0		
PC 端口输出电平	拉幕	1	1	1	0	0	1	1	1	0xE7	D4、D3 点亮
		1	1	0	1	1	0	1	1	0xDB	D5、D2 点亮
		1	0	1	1	1	1	0	1	0xBD	D6、D1 点亮
		0	1	1	1	1	1	1	0	0x7E	D7、D0 点亮
	闭幕	1	0	0	0	0	0	0	1	0x81	D7、D0 熄灭
		1	1	0	0	0	0	1	1	0xC3	D7、D0、D6、D1 熄灭
		1	1	1	0	0	1	1	1	0xE7	D4、D3 亮
		1	1	1	1	1	1	1	1	0xFF	全部熄灭

在 CVAVR 中，要直接实现数据的计算有时不太容易，在此也可以将表中所有的数据取反后再送 PORTC 端口。

> 注意：
> 进行程序设计时，也可以使用两个数组来实现，其程序流程可以参考图 4-10。

源程序

【移位指令实现的源程序】

```
/*****************************************************
  文件名:          拉幕式与闭幕式广告灯.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
  说明:            利用移位指令实现
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
void delay(uint k)
{
   uint    m,n;
   for(m=0;m<k;m++)
      {
         for(n=0;n<1140;n++)
           {;}
      }
}
void main(void)
{
      uchar i,j,k,LED;
      DDRC=0xFF;
     PORTC=0xFF;
```

```
    while(1)
    {
      j=0x08;
      k=0x10;
      for(i=0;i<4;i++)
        {
            LED=j|k;
            PORTC=~LED;
            delay(500);
             j=j<<1;
            k=k>>1;
        }
      LED=0xFF;
      PORTC=~LED;
      delay(500);
      j=0x01;
      k=0x80;
      for(i=0;i<8;i++)
        {
            LED=j|k;
            PORTC=LED;
            delay(500);
            j=(j<<1)+0x01;
            k=(k>>1)+0x80;
          }
      }
}
```

【数组实现的源程序】

```
/*****************************************************
 文件名:          拉幕式与闭幕式广告灯.c
 单片机型号:      ATmega16
 晶振频率:        8.0MHz
 说明:            利用数组实现
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
flash uchar discode1[4]={0x18,0x24,0x42,0x81};
flash uchar discode2[4]={0x7E,0x3C,0x18,0x00};
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
```

```
    }
}
void main(void)
{
    uchar i,j;
    DDRC=0xFF;
    PORTC=0xFF;
    while(1)
      {
        for(i=0;i<4;i++)                    //D0～D7 移位点亮（左移）
            {
              j=discode1[i];
              PORTC=~j;
              delay(500);
            }
        j=0xFF;
        PORTC=~j;
        delay(500);
        for(i=0;i<4;i++)                    //D7～D0 移位点亮（右移）
            {
              j=discode2[i];
              PORTC=~j;
              delay(500);
            }
      }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“拉幕式与闭幕式.COFF”文件，然后在 Proteus ISIS 中打开已创建的拉幕式与闭幕式电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。此时可以看到拉幕式显示效果，首先是 D3、D4 亮，延时片刻后，接着 D2、D5 亮，再依次是 D1、D6 亮，D0、D7 亮。然后为闭幕式显示效果，首先是 D0、D7 灭，延时片刻后，接着 D1、D6 灭，再依次是 D2、D5 灭，然后全部熄灭。全部熄灭后，又重新开始下一轮的拉幕式与闭幕式控制，其仿真效果如图 4-11 所示。



任务 5　复杂广告灯的设计

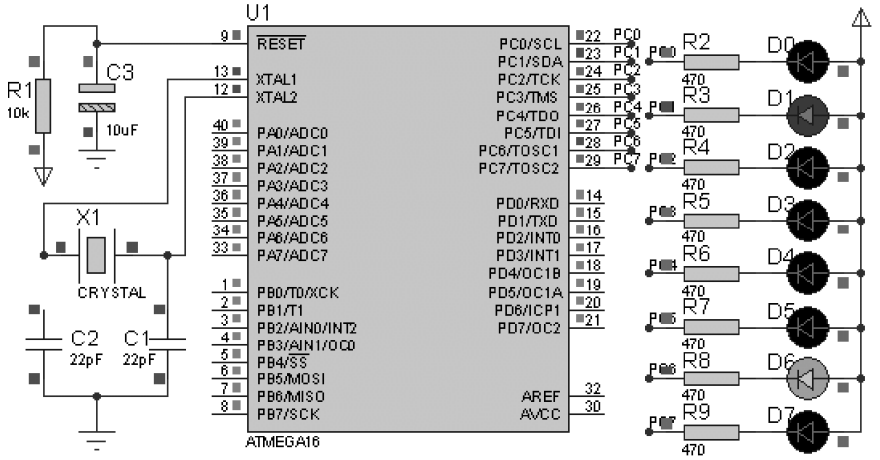
使用单片机 PC 端口实现 8 只 LED 的复杂广告灯控制，要求显示规律为：正向流水→反向流水→隔灯闪烁 3 次→高 4 盏、低 4 盏闪烁 2 次→隔 2 盏闪烁 3 次，再重复循环。



硬件设计

8 只 LED 的复杂广告灯设计所使用的硬件电路与流水广告灯的硬件电路完全相同，因

此可参照图 4-7 绘制复杂广告灯的电路图。



（a）拉幕式

（b）闭幕式

图 4-11　拉幕式与闭幕式广告灯运行仿真效果图

软件设计

对于较复杂的 LED 广告设计时，可以使用数组的方式实现。本设计数组中的显示状态数据见表 4-14。

表 4-14　复杂广告灯显示状态数据

LED	D7	D6	D5	D4	D3	D2	D1	D0	PORTC 输出（十六进制）	功能说明
PC 端口	PC7	PC6	PC5	PC4	PC3	PC2	PC1	PC0		
PC 端口输出电平	1	1	1	1	1	1	1	0	0xFE	正向流水
	1	1	1	1	1	1	0	1	0xFD	
	1	1	1	1	1	0	1	1	0xFB	
	1	1	1	1	0	1	1	1	0xF7	

续表

LED	D7	D6	D5	D4	D3	D2	D1	D0	PORTC 输出（十六进制）	功能说明
PC 端口	PC7	PC6	PC5	PC4	PC3	PC2	PC1	PC0		
PC 端口输出电平	1	1	1	0	1	1	1	1	0xEF	正向流水
	1	1	0	1	1	1	1	1	0xDF	
	1	0	1	1	1	1	1	1	0xBF	
	0	1	1	1	1	1	1	1	0x7F	
	1	0	1	1	1	1	1	1	0xBF	反向流水
	1	1	0	1	1	1	1	1	0xDF	
	1	1	1	0	1	1	1	1	0xEF	
	1	1	1	1	0	1	1	1	0xF7	
	1	1	1	1	1	0	1	1	0xFB	
	1	1	1	1	1	1	0	1	0xFD	
	1	1	1	1	1	1	1	0	0xFE	
	1	1	1	1	1	1	1	1	0xFF	
	1	0	1	0	1	0	1	0	0xAA	隔灯闪烁 3 次
	0	1	0	1	0	1	0	1	0x55	
	1	0	1	0	1	0	1	0	0xAA	
	0	1	0	1	0	1	0	1	0x55	
	1	0	1	0	1	0	1	0	0xAA	
	0	1	0	1	0	1	0	1	0x55	
	1	1	1	1	1	1	1	1	0xFF	
	1	1	1	1	0	0	0	0	0xF0	高 4 盏、低 4 盏闪烁 2 次
	0	0	0	0	1	1	1	1	0x0F	
	1	1	1	1	0	0	0	0	0xF0	
	0	0	0	0	1	1	1	1	0x0F	
	1	1	1	1	1	1	1	1	0xFF	
	0	0	1	1	0	0	1	1	0x33	隔 2 盏闪烁 3 次
	1	1	0	0	1	1	0	0	0xCC	
	0	0	1	1	0	0	1	1	0x33	
	1	1	0	0	1	1	0	0	0xCC	
	0	0	1	1	0	0	1	1	0x33	
	1	1	0	0	1	1	0	0	0xCC	
	1	1	1	1	1	1	1	1	0xFF	

为了更好地观看显示效果，有必要在适当的显示状态下控制 8 只 LED 全部熄灭，所以表 4-7 中添加了 0xFF 的显示状态数据。表 4-15 中的显示状态数据可以直接送给 PC 端口，因此在写入数组中的内容时可以不对其进行取反。表 4-15 中列出了显示状态共有 35 种，在取数组中的内容送给 PORTC 时，需要判断取数次数是否达到 35 次，若达到 35 次，则重新

执行下一次循环，否则继续取数组中的内容。其程序流程图如图 4-12 所示。

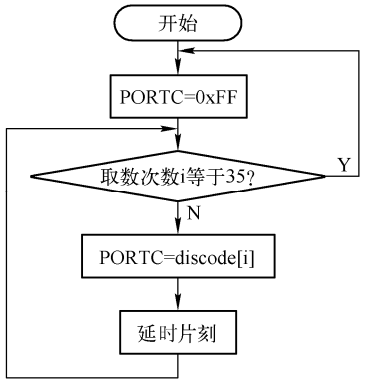


图 4-12　复杂广告灯程序流程图

源程序

```
/****************************************************
文件名:          复杂广告灯.c
单片机型号:      ATmega16
晶振频率:        8.0MHz
****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
flash uchar discode[]={
        0xFE,0xFD,0xFB,0xF7,0xEF,0xDF,0xBF,0x7F,          //正向流水灯
        0xBF,0xDF,0xEF,0xF7,0xFB,0xFD,0xFE,0xFF,          //反向流水灯
        0xAA,0x55,0xAA,0x55,0xAA,0x55,0xFF,               //隔灯闪烁
        0xF0,0x0F,0xF0,0x0F,0xFF,                         //高 4 盏  低 4 盏闪烁
        0x33,0xCC,0x33,0xCC,0x33,0xCC,0xFF};              //隔 2 盏闪烁
void delay(uint k)
{
   uint m,n;
   for(m=0;m<k;m++)
      {
        for(n=0;n<1140;n++)
          {;}
      }
}
void main(void)
{
      uchar i;
      DDRC=0xFF;
      PORTC=0xFF;
      while(1)
         {
```

```
            for(i=0;i<35;i++)
                {
                    PORTC=discode[i];
                     delay(500);
                }
        }
    }
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“复杂广告灯.COFF”文件，然后在 Proteus ISIS 中打开已创建的复杂广告灯电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。先后可以看到 8 只 LED 的显示规律为：正向流水→反向流水→隔灯闪烁 3 次→高 4 盏、低 4 盏闪烁 2 次→隔 2 盏闪烁 3 次，再重复循环，其仿真效果如图 4-13 所示。

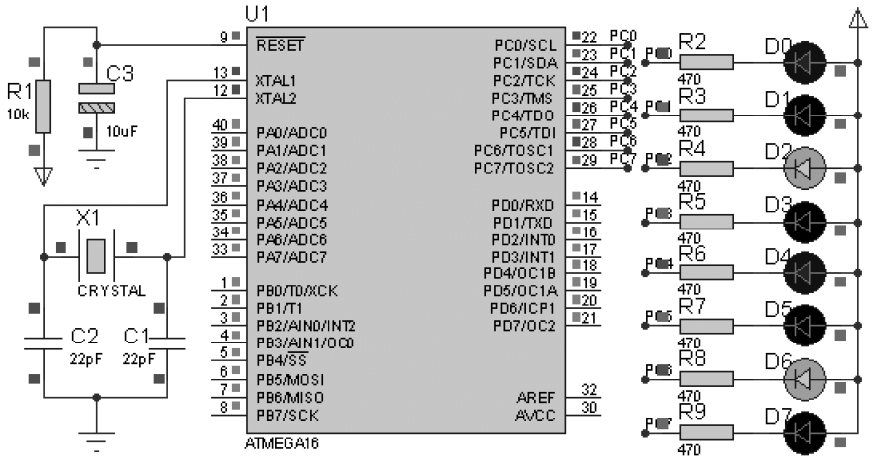


图 4-13　复杂广告灯运行仿真效果图

项目五　按键控制与 LED 数码管显示

〖知识目标〗
☺单片机按键电路的连接;
☺7 段 LED 数码管的结构及工作原理;
☺LED 数码管静态、动态显示控制方式。
〖能力目标〗
☺了解按键的特性及消抖方法;
☺掌握矩阵式键盘控制原理;
☺会运用矩阵键盘实现单片机控制。

在单片机应用系统中，键盘及 LED 数码管是较常见的外围 I/O 设备，常用于人机对话活动的人机通道。在本项目中，主要讲解查询式按键、矩阵键盘、拨号键盘、单个 LED 数码管显示及多个 LED 数码管显示的设计。



任务 1　按键控制原理认知

键盘是一组按键的集合，它是最常用的单片机输入设备。操作人员可以通过键盘输入数据或命令，实现简单的人机通信。

键盘按其结构形式可分为编码键盘和非编码键盘两种方式。编码键盘通过硬件的方法产生键码，能自动识别按下的键并产生相应的键码值，以并行或串行的方式发送给 CPU，其接口简单，响应速度快，但需专用的硬件电路；非编码键盘通过软件的方法产生键码，它不需专用的硬件电路，结构简单成本低廉，但响应速度没有编码键盘快。为了减少电路的复杂程度，节省单片机的 I/O 端口，因此非编码键盘在单片机应用系统中的使用非常广泛。

1．去抖动

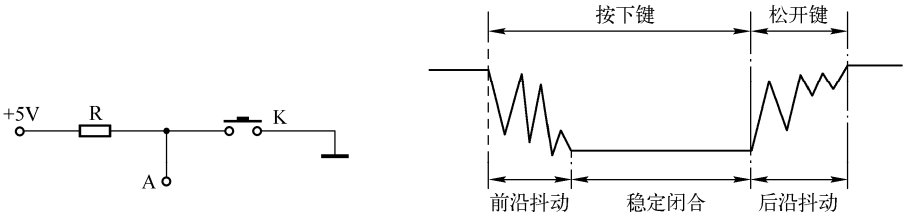
键盘是由按键构成的，键的闭合与否通常用高、低电平来进行检测，键闭合时，该键为低电平；键断开时，该键为高电平。在默认状态下（即平时），按键的两个触点处于断开状态，按下键时它们才闭合。

按键的闭合与断开都是利用其机械弹性，由于机械弹性的作用，按键（K）在闭合与断开的瞬间均有抖动过程，如图 5-1 所示。抖动时间的长短由按键的机械特性决定，一般为 5～10ms。

在触点抖动期间检测按键的通与断的状态，可能导致判断出错，即按键一次按下或释放被错误地认为是多次操作，这种情况是不允许出现的。为了克服按键触点机械抖动所致的错误，使 CPU 对键的一次闭合仅作一次键输入处理，必须采取去抖动措施。

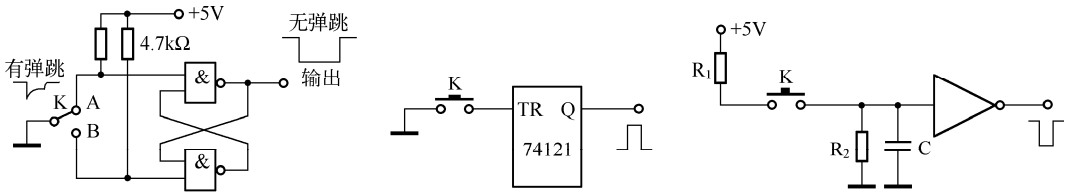
去抖动方法分为硬件方法和软件方法两种。例如，采用 RS 触发器（双稳态触发器）构

成的去抖电路、单稳态触发器构成的去抖电路、滤波电路防抖电路，这些是硬件去抖法，如图 5-2 所示。在图 5-2（a）中当按键 K 未按下时，输出为 1；当 K 按下时，输出为 0。此时即使用按键的机械性能，使按键因弹性抖动而产生瞬时断开（抖动跳开 B），只要按键不返回原始状态 A，双稳态电路的状态不改变，输出保持为 0，不会产生抖动的波形。也就是说，即使 B 点的电压波形是抖动的，但经双稳态电路后，其输出为正规的矩形波。



（a）键输入　　（b）键抖动

图 5-1　按键时的抖动



（a）双稳态去抖动电路　（b）单稳态去抖动电路　（c）滤波去抖动电路

图 5-2　硬件去抖动

软件去抖动方法就是检测到有键按下时，执行一个 10～20ms 的延时子程序后，再确认该键是否仍保持闭合状态，若仍闭合则确认为此键被按下，消除了抖动影响。软件去抖动流程图如图 5-3 所示。

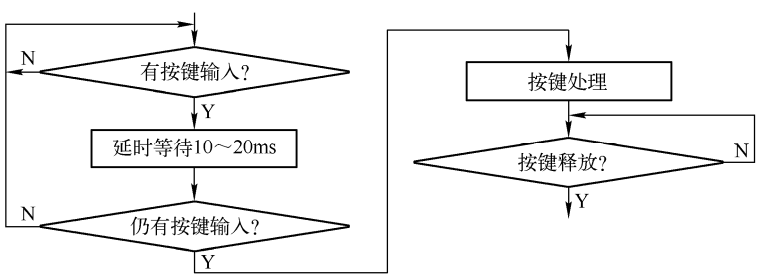


图 5-3　软件去抖动流程图

2．键盘结构

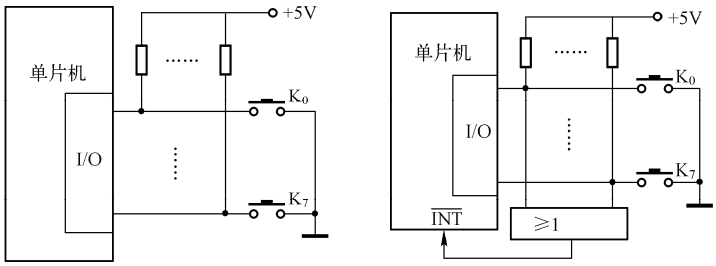
键盘可以分为独立连接式和矩阵式两类，每一类按其译码方法又可以分为编码及非编码两种类型，本书所使用的键盘均为非编码键盘。

独立连接式键盘是直接用 I/O 口线构成的单个按键电路，其特点是每个按键接一根输入线，占用一根 I/O 口线，各键的工作状态互不影响，如图 5-4 所示。

独立连接式键盘电路配置灵活，软件结构简单，但每个按键必须占用一根 I/O 口线，所以在按键较多时，I/O 口线浪费较大，不宜采用。

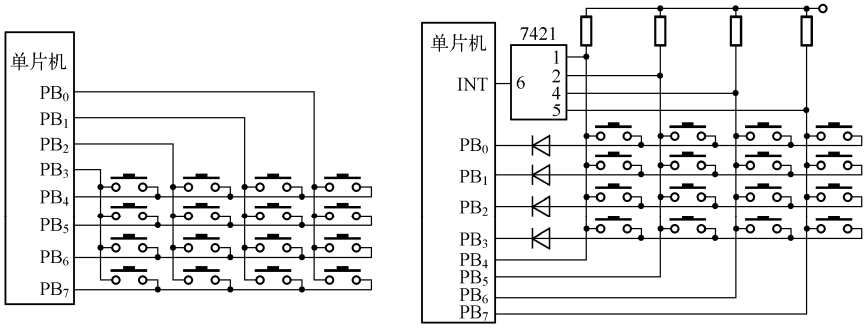
矩阵式键盘又称行列式键盘。用 I/O 口线组成行、列结构，行列线分别连在按键开关的两端，列线通过上拉电阻接至电源，使无键按下时列线处于高电平状态。按键设置在行、列

线的交叉点上。按键按下时，行线与列线发生短路。矩阵式键盘具有占用 I/O 口线较少，软件结构较复杂等特点，适用于按键较多的场合。用 3×3 的行列结构可构成 9 个键的键盘，用 4×4 的行列结构可构成 16 个键的键盘，如图 5-5 所示。



（a）查询方式　　（b）中断方式

图 5-4　独立连接式键盘电路



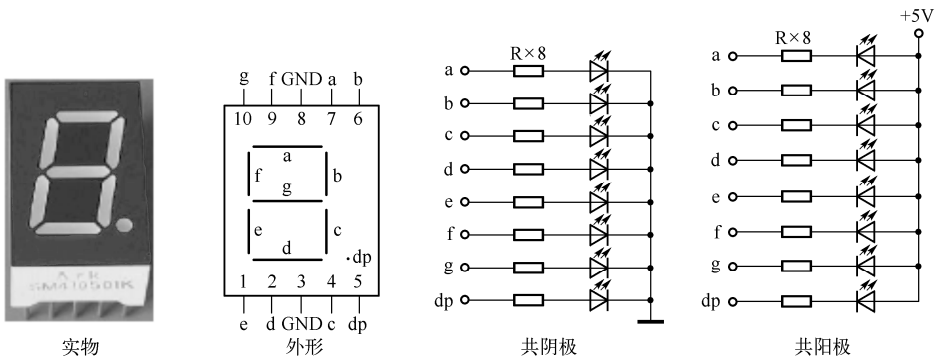
（a）扫描法　　（b）中断法

图 5-5　矩阵式键盘

任务 2　LED 数码管显示原理认知

1．LED 数码管结构及字形代码

通常使用的 LED 数码管是 7 段 LED，它是由 7 个 LED 组成。这 7 个 LEDa～g 呈日字形排列，其结构及连接如图 5-6 所示。当某一 LED 导通时，相应地点亮某一点或某一段笔画，通过 LED 不同的亮暗组合形成不同的数字、字母及其他符号。



实物　　外形　　共阴极　　共阳极

图 5-6　LED 数码管结构及连接

〖**LED 数码管中 LED 的两种接法**〗

☺ 所有 LED 的阳极连接在一起，这种连接方法称为共阳极接法。

☺ 所有 LED 的阴极连接在一起，这种连接方法称为共阴极接法。

共阳极的 LED 为低电平时，对应的段码被点亮；共阴极的 LED 为高电平时，对应段码被点亮。一般共阴极可以不外接电阻，但共阳极中的 LED 一定要外接电阻。

LED 数码管的 LED 亮暗组合实质上就是不同电平的组合，也就是为 LED 数码管提供不同的代码，这些代码称为字形代码，即段码。7 段 LED 加上 1 个小数点 dp 共计 8 段，字形代码与这 8 段的关系如下：

数据字	D7	D6	D5	D4	D3	D2	D1	D0
LED 段	dp	g	f	e	d	c	b	a

字形代码与十六进制数的对应关系见表 5-1。从表中可以看出，共阴极与共阳极的字形代码互为补数。

表 5-1　字形代码与十六进制数对应关系

字　符	dp	g	f	e	d	c	b	a	段码（共阴极）	段码（共阳极）
0	0	0	1	1	1	1	1	1	3FH	C0H
1	0	0	0	0	0	1	1	0	06H	F9H
2	0	1	0	1	1	0	1	1	5BH	A4H
3	0	1	0	0	1	1	1	1	4FH	B0H
4	0	1	1	0	0	1	1	0	66H	99H
5	0	1	1	0	1	1	0	1	6DH	92H
6	0	1	1	1	1	1	0	1	7DH	82H
7	0	0	0	0	0	1	1	1	07H	F8H
8	0	1	1	1	1	1	1	1	7FH	80H
9	0	1	1	0	1	1	1	1	6FH	90H
A	0	1	1	1	0	1	1	1	77H	88H
B	0	1	1	1	1	1	0	0	7CH	83H
C	0	0	1	1	1	0	0	1	39H	C6H
D	0	1	0	1	1	1	1	0	5EH	A1H
E	0	1	1	1	1	0	0	1	79H	86H
F	0	1	1	1	0	0	0	1	71H	8EH
—	0	1	0	0	0	0	0	0	40H	BFH
.	1	0	0	0	0	0	0	0	80H	7FH
熄灭	0	0	0	0	0	0	0	0	00H	FFH

2．LED 数码管的显示方式

在单片机应用系统中一般需使用多个 LED 显示器，多个 LED 显示器是由 *n* 根位选线和 8×*n* 根段选线连接在一起的。根据显示方式不同，位选线与段选线的连接方法也不相同。段选线控制字符选择，位选线控制显示位的亮或暗。其连接方法如图 5-7 所示。

LED 显示器有静态显示和动态显示两种工作方式。

【静态显示】静态显示就是当 LED 数码管要显示某一个字符时，相应的 LED 恒定地导通或截止。例如，LED 数码管要显示“0”时，a、b、c、d、e、f 导通，g、dp 截止。单片机将所要显示的数据送出去后就不需再管它，直到下一次显示数据需更新时再传送一次数据即可，显示数据稳定，占用 CPU 时间少。但这种显示方式，每一位都需要一个 8 位输出口控制，所以占用硬件多，如果单片机系统中有 *n* 个 LED 数码管时，需 8×*n* 根 I/O 口线，所占用的 I/O 资源较多，需进行扩展。在实际应用中，使用 74HC595 芯片可以方便地组成静态显示接口，该显示接口所占用的 I/O 资源较少。

74HC595 是一款漏极开路输出的 CMOS 移位寄存器，其内部结构框图如图 5-8 所示。从图中可以看出，74HC595 内部有一个带有进位位的 8 位移位寄存器、一个存储寄存器和一个三态输出控制器。

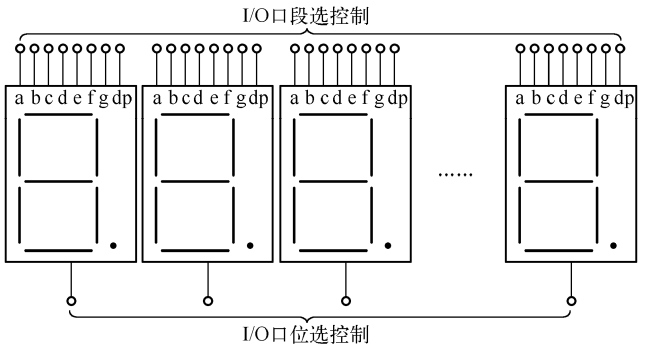


图 5-7　*n* 个 LED 显示器的连接

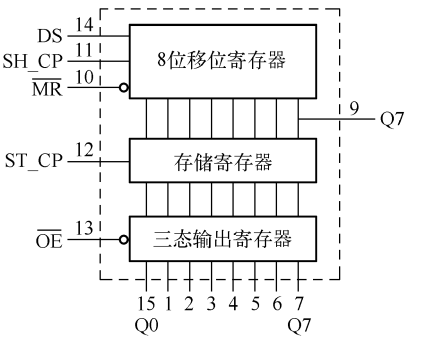


图 5-8　74HC595 的内部结构框图

当时钟端 SH_CP（第 11 脚）有时钟脉冲时，移位寄存器将串行输入端 DS（第 14 脚）的数据转换成并行输出。在串行数据开始输入前，将 ST_CP（第 12 脚）清零，移位寄存器的输出不会被送入存储寄存器，在 8 位数据全部送完后，将 ST_CP 引脚置 1，才会将新的数据送入存储寄存器中。存储寄存器经过三态控制器缓冲后，对外输出。这样，在整个数据传输期间，74HC595 的输出端数据始终保持稳定不变。8 位移位寄存器的进位位单独引出（即第 9 脚），可以方便地进行级联，以便将多片 74HC595 串接起来使用。

74HC595 的逻辑功能见表 5-3。在表中，“H”表示高电平；“L”表示低电平；“↑”表示上升沿；“↓”表示下降沿；“X”表示无关紧要，高或低电平均不影响。

表 5-2　74HC595 的逻辑功能表

输入引脚					功　能
DS	SH_CP	$\overline{MR}$	ST_CP	$\overline{OE}$	
X	X	X	X	H	禁止 Q0～Q7 输出
X	X	X	X	L	允许 Q0～Q7 输出
X	X	L	X	X	清除内部移位寄存器
L	↑	H	X	X	移位寄存器的首位变低，其余各位依次前移
H	↑	H	X	X	移位寄存器的首位变高，其余各位依次前移
X	↓	H	X	X	移位寄存器的内容不发生变化
X	X	X	↑	X	移位寄存器中的数据送入存储寄存器
X	X	X	↓	X	存储寄存器的输出不发生变化

图 5-9 所示的是以 74HC595 组成的静态显示接口电路图，通过 4 片 74HC595 作为 4 位 LED 显示器的静态显示接口。其中，第 1 片 74HC595（U2）的串行数据输入端（DS）接到单片机的任意一个 I/O 端口，后面的 74HC595 芯片的 DS 端则接到前一片 74HC595 的 Q7′端（第 9 脚）。所有 74HC595 芯片的 SH_CP 端并联接到单片机的任意一个 I/O 端口。ST_CP 是锁存使用端，当该引脚上有上升沿且其他条件符合时，移位寄存器中的内容被送入存储寄存器。

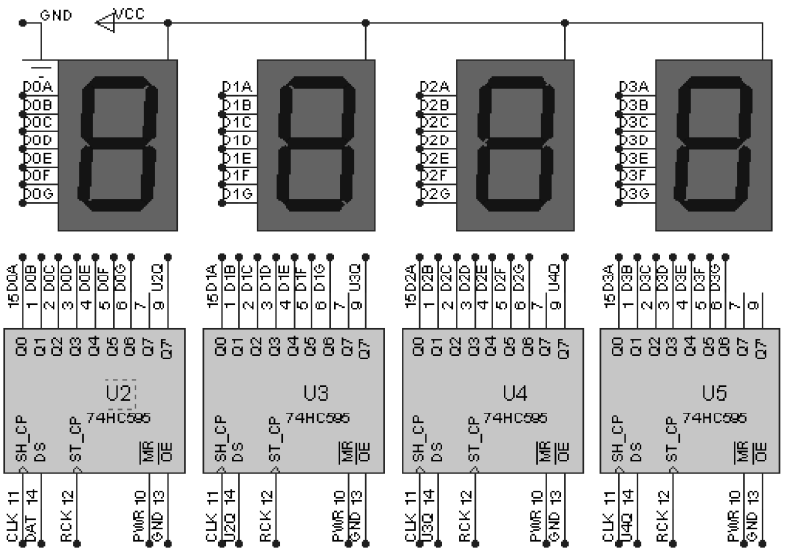


图 5-9　4 位 LED 数码管静态显示接口

【动态显示】动态显示就是逐位地轮流点亮各位显示器，对于每一位 LED 数码管来说，每隔一段时间点亮一次，即 CPU 需时刻对显示器进行刷新，显示数据有闪烁感，占用 CPU 时间较多。且显示器的点亮既跟点亮时的导通电流有关，也跟点亮时间和间隔时间的比例有关。调整电流和时间的参数，可实现亮度较高、较稳定的显示。但是若显示器的位数不大于 8 位时，只需两个 8 位 I/O 端口即可。

不管是静态显示还是动态显示，既可以由普通的 I/O 端口驱动，也可以由特殊 I/O 端口通过扩展外围驱动芯片来驱动。在本项目中，主要讲述 LED 数码管动态显示，对于由特殊 I/O 端口通过扩展外围驱动芯片来驱动 LED 的知识，将在项目十二和项目十三中进行讲解。除此之外，本书中的多位 LED 显示是采用普通 I/O 端口驱动的动态显示。

3．LED 数码管的识别与检测方法

LED 数码管的识别与检测可以使用干电池检测及万用表检测这两种方法进行。

【干电池检测法】取两节普通 1.5V 干电池串联起来形成 3V 电压源，并串联一个 100Ω、1/8W 的限流电阻，以防止电流过大烧坏被测 LED 数码管。将 3V 电压的负极引线接在被测数码管的公共阴极上，正极引线依次移动接触各笔段电极（a～h 脚）。当正极引线接触到 LED 数码管的某一段码电极时，对应段码就发光显示。用这种方法可以快速测出数码管是否有断笔（某一段码不能显示）或连笔（某些段码连在一起），并且可相对比较出不同的段码发光强弱是否一致。若检测共阳极数码管，只需将电池的正、负极引线对调一下，方法同上。

被测数码管的各笔段电极（a～h 脚）全部短接起来，再接通测试用干电池 3V 电压源，

则可使被测数码管实现全段码发光。对于质量保证的数码管，其发光颜色应该均匀，并且无段码残缺及局部变色等。

如果不清楚被测数码管的是共阳极还是共阴极和引脚排序，可从被测数码管的左边第 1 脚开始，逆时针方向依次逐脚测试各引脚，使各段码分别发光，即可测绘出该数码管的引脚排列和内部接线。

> 注意：
>
> 测试时，只要某一段码发光，就说明被测的两个引脚中有一个是公共脚，假定某一脚是公共脚不动，变动另一测试脚，如果另一个段码发光，说明假定正确。这样根据公共脚所接电源的极性，可判断出被测数码管是共阳极还是共阴极。显然，公共引脚如果接电池正极，则被测数码管为共阳极接法；公共引脚如果接电池负极，则被测数码管应为共阴极接法。接下来测试剩余各引脚，即可很快确定出所对应的段码来。

【万用表检测法】这里以指针式万用表为例，说明具体检测方法：首先将指针式万用表拨至 *R*×10k 挡。由于 LED 数码管内部的 LED 正向导通电压一般不小于 1.8V，所以万用表的电阻挡应置于内部电池电压是 15V（或 9V）的 *R*×10k 挡，而不应置于内部电池电压是 1.5V 的 *R*×100 或 *R*×1k 挡，否则无法正常测量 LED 的正、反向电阻。然后，进行检测。在测量共阴极数码管时，万用表红表笔（注意：红表笔接表内电池负极、黑表笔接表内电池正极）应接数码管的“–”公共端，黑表笔则分别去接各段码电极（a～g、dp 脚）；对于共阳极的数码管，黑表笔应接数码管的“+”公共端，红表笔则分别去接 a～g、dp 脚。正常情况下，万用表的指针应该偏转（一般示数在 100kΩ 以内），说明对应段码的 LED 导通，同时对应段码会发光。若测到某个引脚时，万用表指针不偏转，所对应的段码也不发光，则说明被测段码的 LED 已经开路损坏。与干电池检测法一样，采用万用表检测法也可对不清楚结构类型和引脚排序的数码管进行快速检测。

4．LED 数码管编码器的使用

在编写程序过程中，有时需要编写 LED 数码管的显示代码，如果用户忘记了字形代码时，可使用数码管编码器来生成 7 段 LED 显示驱动码，如图 5-10 所示。有了这个软件后，再也不用那么辛苦地记住字形代码。只要在该软件中进行相关设置，就可以马上生成 C 语言或汇编的段码表。

该软件的使用比较简单，用户首先根据目标板上 LED 数码管的类型，在此软件中选择数码管的极性，然后根据 LED 引脚与单片机的连接方式来确定段码表。例如，用户使用的 LED 数码管为共阳极，LED 数码管的段码 a 与单片机的 RD_0 连接，段码 b 与 RD_1 连接……段码 h 与 RD_7 连接时，可参照图 5-11 所示进行相关设置。设置好后，单击“编码”按钮，就可以生成字形代码，用户直接复制这些字形代码即可。

任务 3　单个 LED 数码管显示设计

使用单片机 PC 端口作为输出端口，外接 LED 数码管，编写程序，使数码管循环显示从 0～9 的加 1 计数。



图 5-10　数码管编码器

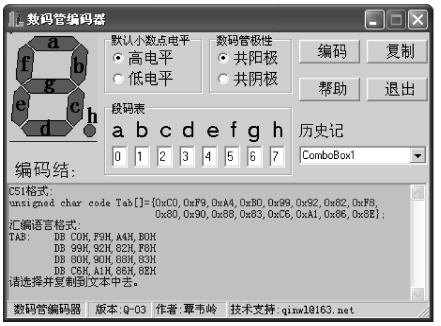


图 5-11　数码管编码器的设置



硬件设计

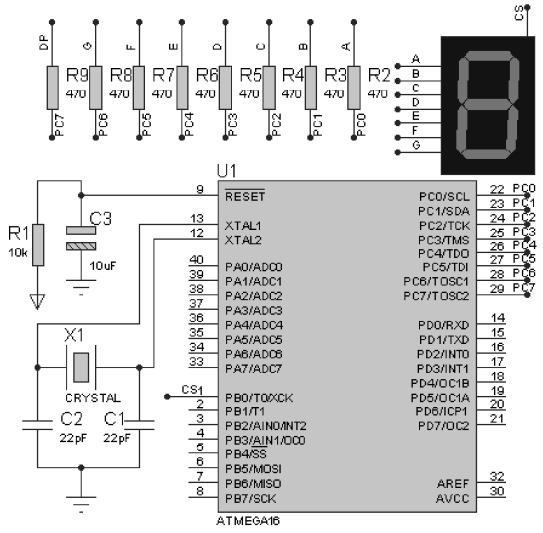
单片机 I/O 端口控制 LED 数码管进行显示时，其限流电阻的要求与项目四中 LED 的控制要求基本类似，在此不再具体叙述。

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 5-3 所列的元器件，并完成图 5-12 所示的硬件电路图设计。

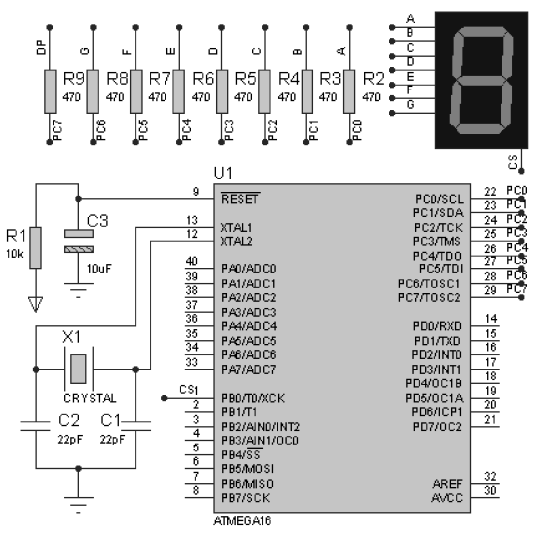
> 注意:
> 在 Proteus 中只要绘制出图 5-12（a）或（b）中一个电路图即可。

表 5-3　单个 LED 数码管显示设计所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	共阳 LED 7SEG-COM-AN-GRN	晶振 CRYSTAL 8MHz
电阻 RES	电解电容 CAP-ELEC	共阴 LED 7SEG-COM-CAT-BLUE	



（a）共阳 LED 数码管显示电路



（b）共阴 LED 数码管显示电路

图 5-12　单个 LED 数码管显示电路

软件设计

单个 LED 数码的显示，采用静态显示工作原理。要使 LED 数码管显示 0～9，则 PORTC 对应输出 7 段数码管数字显示相应的字形代码即可。通常字形代码放在创建的字库中，该字库的定义也就是一个数组的定义。

> 注意:
> 在定义数组 LED_code[]时，由于共阳极 LED 数码管和共阴极 LED 数码管显示 0～9 的字形代码不同，因此数组的元素也不相同，但其编程思路是相同的。

使用 C 语言进行编程时，实现任务的思路是，程序开始时，给数组元素的变量赋初始值 0，并将数组中第 1 个元素送 PC 端口，延时片刻，将变量 i 加 1，并判断是否已经读取到第 10 个元素，如果已经读完，则对变量 i 重新赋值；如果没有，则继续读取数组中第 i 个元素送 PC 端口，依次循环。其程序流程如图 5-13 所示。

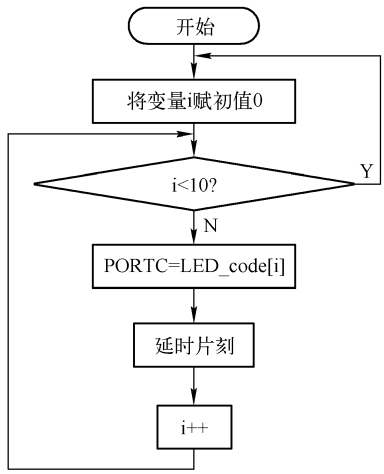


图 5-13　单个 LED 数码管显示程序流程图

源程序

【实现共阳极 LED 数码管显示的源程序】

```
/*************************************************
  文件名:          单个 LED 数码管显示.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
  说明:            共阳极
*************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
flash uchar LED_code[16]={
          0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,        //共阳极 LED   0～F 的段码
          0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E};
void delay(uint k)                                       //约 1ms 延时函数
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void port_init()                                         //端口初始化
{
  PORTB=0xFE;                                            //PB 端口初始值设置
  DDRB=0xFF;                                             //PB 端口配置为输出
```

```
    PORTC=0xFF;                                          //PC 端口初始值设置
    DDRC=0xFF;                                           //PC 端口配置为输出
}
void main(void)
{
    uchar i;
    port_init();
    PORTB=0x01;                                          //CS 有效
    while(1)
      {
        for(i=0;i<10;i++)
          {
            PORTC=LED_code[i];
            delay(500);
          }
      }
}
```

【实现共阴极 LED 数码管显示的源程序】

```
/*****************************************************
文件名:          单个 LED 数码管显示.c
单片机型号:      ATmega16
晶振频率:        8.0MHz
说明:            共阴极
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
flash uchar LED_code[16]={
          0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,       //共阴极 LED   0～F 的段码
          0x7F,0x6F,0x77,0x7C,0x39,0x5E,0x79,0x71};
void delay(uint k)                                       //约 1ms 延时函数
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void port_init()                                         //端口初始化
{
  PORTB=0xFF;                                            //PB 端口初始值设置
  DDRB=0xFF;                                             //PB 端口配置为输出
  PORTC=0xFF;                                            //PC 端口初始值设置
  DDRC=0xFF;                                             //PC 端口配置为输出
}
```

```
void main(void)
{
    uchar i;
    port_init();
    PORTB=0x00;       //CS 有效
    while(1)
      {
        for(i=0;i<10;i++)
          {
            PORTC=LED_code[i];
            delay(500);
          }
      }
}
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“单个 LED 数码管显示.COFF”文件，然后在 Proteus ISIS 中打开已创建的单个 LED 数码管显示电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。此时看到 LED 数码管依次显示数字 0～9，显示 9 后又重新从 0 开始，重复循环，仿真效果如图 5-14 所示。

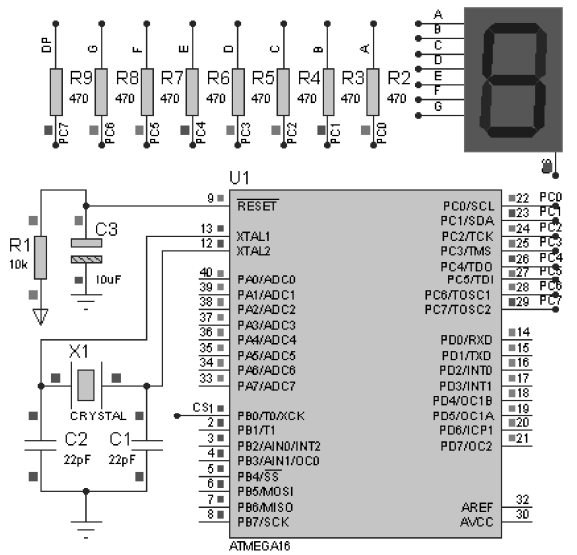


图 5-14　单个 LED 数码管显示仿真效果图

任务 4　多个 LED 数码管显示设计

使用单片机 PC 和 PB 端口作为输出口，外接一个 8 位 LED 数码管，编写程序，使数码管显示“872AF635”。

硬件设计

根据 LED 数码管显示方式得知，单片机控制 8 位 LED 数码管进行显示时，通常采用动态显示方式。使用单片机 PC 和 PB 端口控制 8 位 LED 数码管显示相应内容时，在硬件设计中应该考虑 PC 和 PB 端口的 I/O 驱动能力。通常 PC 端口控制每位 LED 数码管笔段的亮或灭；PB 端口实现 LED 数码管的片选控制。所以在硬件电路设计中，PC 端口需接限流电阻，再与 LED 数码管的段码连接，PB 端口通过电流驱动芯片（如 74LS245）或三极管再与 LED 数码管的片选端连接。

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 5-4 所列的元器件，并完成图 5-15 所示的硬件电路图设计。

表 5-4　多个 LED 数码管显示设计所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	晶振 CRYSTAL 8MHz	电解电容 CAP-ELEC
电阻 RES	三极管 NPN	数码管 7SEG-MPX8-CA-BLUE	

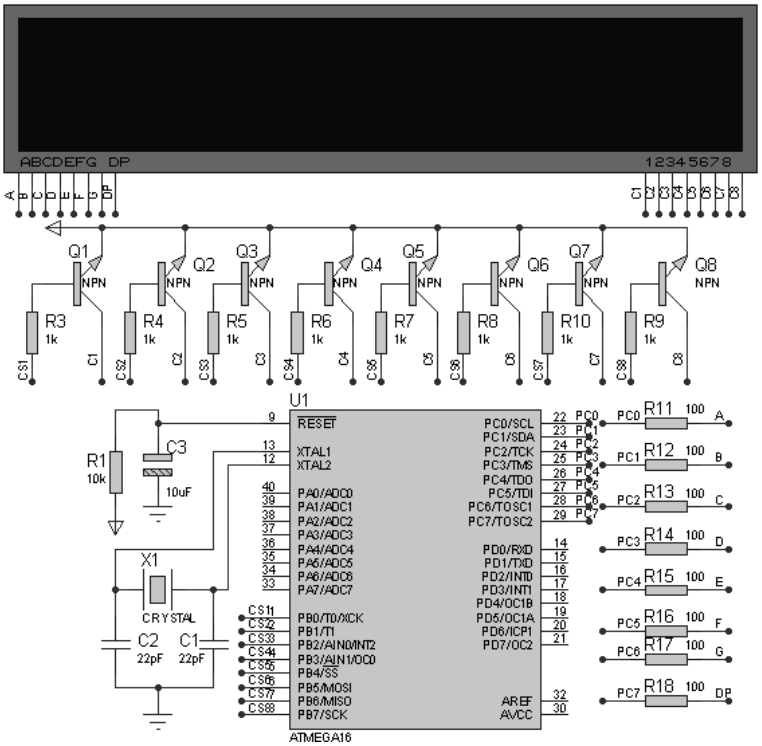
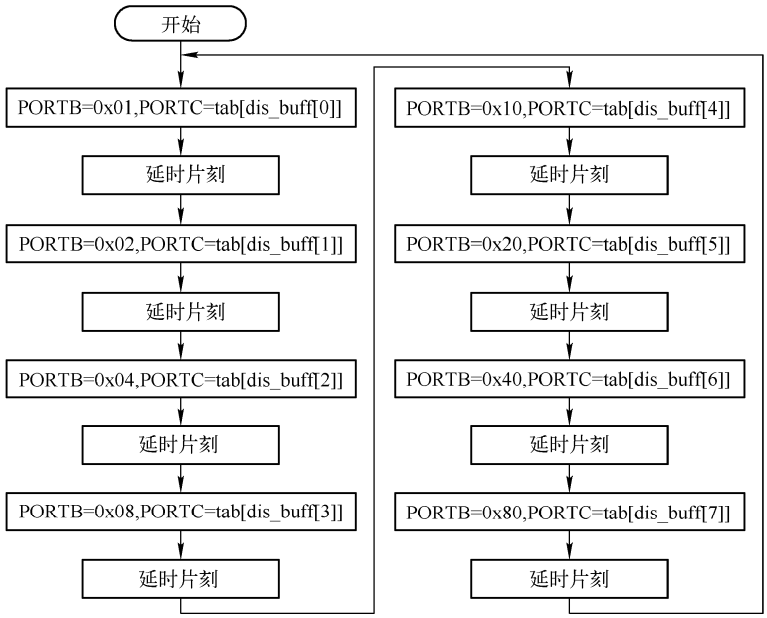


图 5-15　多个 LED 数码管显示电路

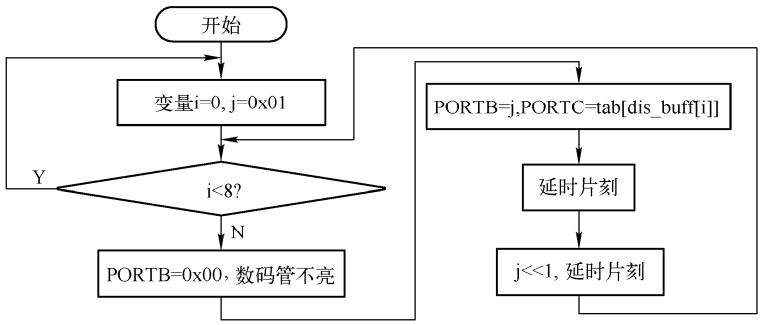
软件设计

多个 LED 数码管要显示“872AF635”，也需要将所显示的字形代码放在所创建的数组中。在此定义数组时，与单个 LED 数码管的数组定义有所不同，应该先建立一个 0～F 的数组 tab[]，然后再建立一个数组 dis_buff[]指定所显示的内容。多个 LED 数码管动态显示

时，根据数码管动态显示方式可知，PB0～PB7 依次输出高电平，以控制对应的数码管点亮选通，同时 PC 端口输出相应的字形代码，使已选通点亮的 LED 数码管能显示相应的内容。要实现这些控制可以采用两种方式实现，即顺序控制及循环赋值控制。其程序流程图如图 5-16 所示。



（a）顺序控制实现多个LED数码管显示程序流程图



（b）循环赋值实现多个LED数码管显示程序流程图

图 5-16　多个 LED 数码管显示程序控制流程图

源程序

【顺序控制实现多个 LED 数码管显示的源程序】

```
/*****************************************************
  文件名:          多个 LED 数码管显示.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
  说明:            8 位一体共阳极, 顺序控制实现
*****************************************************/
```

```
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
flash uchar tab[17]={
            0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,                    //共阳极 LED  0～F 的段码
            0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E,0xBF};              //"0xBF"表示"-"
flash uchar dis_buff[8]={0x08,0x07,0x02,0x0a,0x0f,0x06,0x3,0x05};       //显示值
void delay(uint k)                                                      //约 1ms 延时函数
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void port_init()                                                        //端口初始化
{
  PORTB=0x00;                                                           //PB 端口初始值设置
  DDRB=0xFF;                                                            //PB 端口配置为输出
  PORTC=0xFF;                                                           //PC 端口初始值设置
  DDRC=0xFF;                                                            //PC 端口配置为输出
}
void display(void)
{
  PORTB=0x01;
  PORTC=tab[dis_buff[0]];
  delay(1);
  PORTB=0x02;
  PORTC=tab[dis_buff[1]];
  delay(1);
  PORTB=0x04;
  PORTC=tab[dis_buff[2]];
  delay(1);
  PORTB=0x08;
  PORTC=tab[dis_buff[3]];
  delay(1);
  PORTB=0x10;
  PORTC=tab[dis_buff[4]];
  delay(1);
  PORTB=0x20;
  PORTC=tab[dis_buff[5]];
  delay(1);
  PORTB=0x40;
  PORTC=tab[dis_buff[6]];
  delay(1);
  PORTB=0x80;
  PORTC=tab[dis_buff[7]];
```

```
    delay(1);
}
void main(void)
{
     port_init();
     while(1)
        {
          display();
        }
}
```

【循环赋值控制实现多个 LED 数码管显示的源程序】

```
/*****************************************************
 文件名:          多个 LED 数码管显示.c
 单片机型号:      ATmega16
 晶振频率:        8.0MHz
 说明:            8 位一体共阳极，循环赋值控制实现
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
flash uchar tab[17]={
            0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,                  //共阳极 LED  0～F 的段码
            0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E,0xBF};            //"0xBF"表示"-"
flash uchar dis_buff[8]={0x08,0x07,0x02,0x0a,0x0f,0x06,0x3,0x05};     //显示值
void delay(uint k)                                                    //约 1ms 延时函数
{
   uint m,n;
   for(m=0;m<k;m++)
      {
         for(n=0;n<1140;n++)
           {;}
      }
}
void port_init()                                                      //端口初始化
{
   PORTB=0x00;                                                        //PB 口初始值设置
   DDRB=0xFF;                                                         //PB 口配置为输出
   PORTC=0xFF;                                                        //PC 口初始值设置
   DDRC=0xFF;                                                         //PC 口配置为输出
}
void display(void)
{
   uchar i,j;
   j=0x01;
   for(i=0;i<8;i++)
```

```
        {
          PORTB=0x00;
          PORTC=tab[dis_buff[i]];
          PORTB=j;
          j=j<<1;
          delay(1);
        }
    }
    void main(void)
    {
      port_init();
      while(1)
        {
          display();
        }
    }
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“多个 LED 数码管显示.COFF”文件，然后在 Proteus ISIS 中打开已创建的多个 LED 数码管显示电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。此时看到 8 个 LED 数码管显示的内容，如图 5-17 所示。

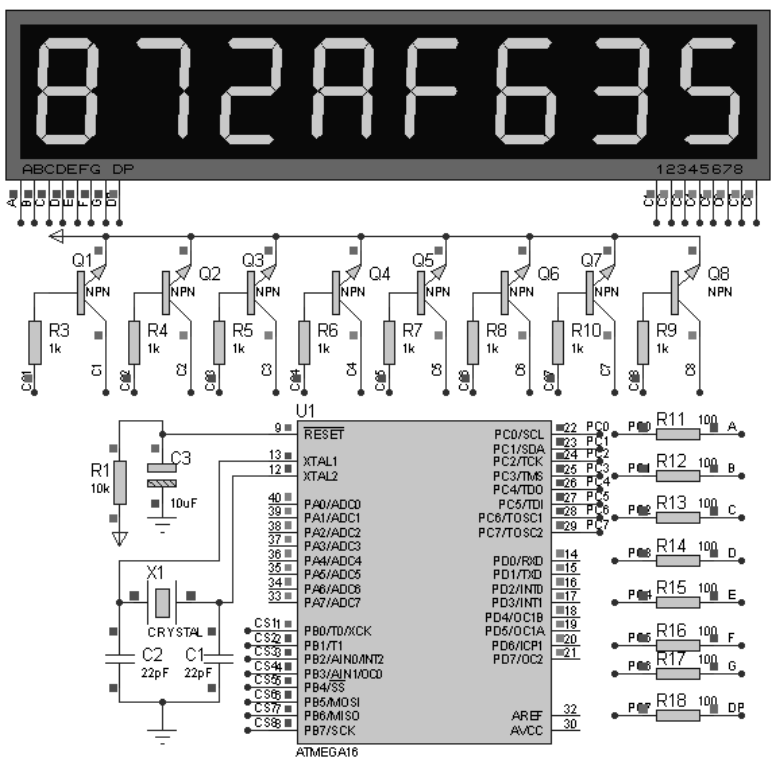


图 5-17　多个 LED 数码管显示仿真效果图

由于人的视觉暂留效应，如果将 delay(1)指令中的数字改大时，则会看到 LED 数码管显示不太稳定，有一定的闪烁感。更改数组 dis_buff[]中数字，显示内容也会发生相应改变。

任务 5　条形 LED 花样选择显示

单片机 PA 端口外接 4 按键（K1～K4）的拨码开关，PC 端口外接条形 LED，要求拨码开关 K1 接通时，条形 LED 进行右移流水显示；K2 接通时，条形 LED 进行左移流水显示；K3 接通时，条形 LED 高、低 4 位互闪两次显示；K4 接通时，条形 LED 隔灯闪烁 2 次。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 5-5 所列的元器件，并完成如图 5-18 所示的硬件电路图设计。

表 5-5　条形 LEDLED 花样选择显示所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	晶振 CRYSTAL 8MHz	电解电容 CAP-ELEC
电阻 RES	拨码开关 DIPSW_4	条形 LED LED-BARGRAPH -GRN	

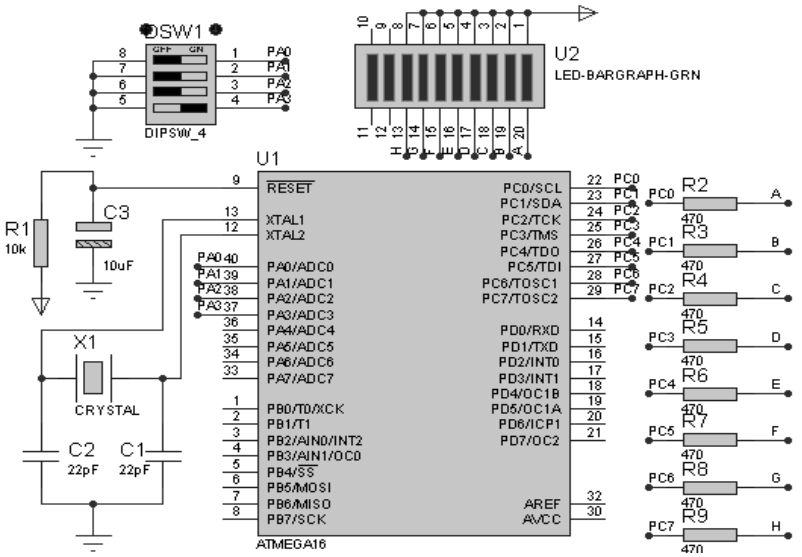


图 5-18　条形 LED 花样选择显示电路

软件设计

条形 LED 的显示花样由拨码开关进行选择控制，拨码开关的 4 个按键分别对应 4 种不同的显示花样，因此可以将这 4 种花样状态通过建立 4 个数组来实现。花样的选择，在程序中可以使用 switch(PINA)语句来控制。其程序流程图如图 5-19 所示。

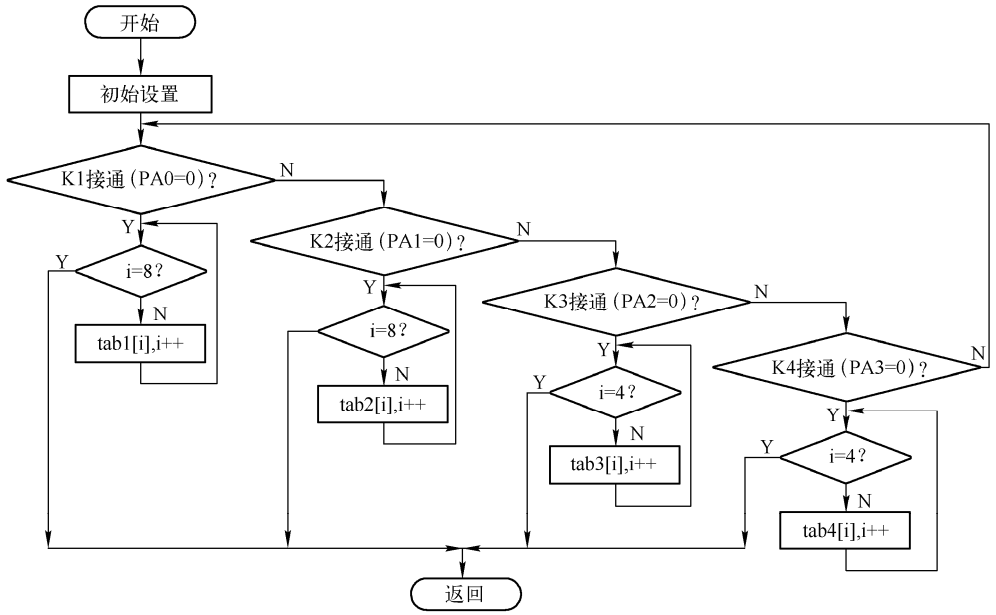


图 5-19　条形 LED 花样选择显示程序流程图

源程序

```
/*****************************************************
  文件名:          条形 LED 花样选择显示.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
flash uchar tab1[]={0x7F,0xBF,0xDF,0xEF, 0xF7,0xFB,0xFD,0xFE};      //右移流水
flash uchar tab2[]={0xFE,0xFD,0xFB,0xF7, 0xEF,0xDF,0xBF,0x7F};      //左移流水
flash uchar tab3[]={0x0F,0xF0,0x0F,0xF0};                           //高低互闪 2 次
flash uchar tab4[]={0xAA,0x55,0xAA,0x55};                           //隔灯闪烁 2 次
void delay(uint k)                                                  //约 1ms 延时函数
{
   uint m,n;
   for(m=0;m<k;m++)
      {
         for(n=0;n<1140;n++)
           {;}
      }
}
void port_init()                                                    //端口初始化
{
   PORTA=0xFF;                                                      //PA 端口初始值设置
   DDRA=0x00;                                                       //PA 端口配置为输入
   PORTC=0xFF;                                                      //PC 端口初始值设置
```

```
    DDRC=0xFF;                                          //PC 端口配置为输出
}
void main(void)
{
    uchar j;
    port_init();
    while(1)
     {
        switch(PINA)
        {
           case 0xFE:                                   //PA0 接通
              {
                 for(j=0;j<8;j++)
                   {
                     PORTC=tab1[j];
                      delay(500);
                   }
                 PORTC=0xFF;
                 break;
              }
           case    0xFD:                                //PA1 接通
               {
                  for(j=0;j<8;j++)
                    {
                      PORTC=tab2[j];
                       delay(500);
                    }
                  PORTC=0xFF;
                  break;
               }
           case    0xFB:                                //PA2 接通
               {
                  for(j=0;j<4;j++)
                    {
                      PORTC=tab3[j];
                       delay(500);
                    }
                  PORTC=0xFF;
                  break;
              }
            case    0xF7:                               //PA3 接通
              {
                 for(j=0;j<4;j++)
                   {
                     PORTC=tab4[j];
                      delay(500);
                   }
                 PORTC=0xFF;
```

```
                break;
            }
        }
    }
}
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“条形 LED 花样选择显示.COFF”文件，然后在 Proteus ISIS 中打开已创建的条形 LED 花样选择显示电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。此时拨动拨码开关，会看到条形 LED 显示相应的花样，其仿真效果如图 5-20 所示。

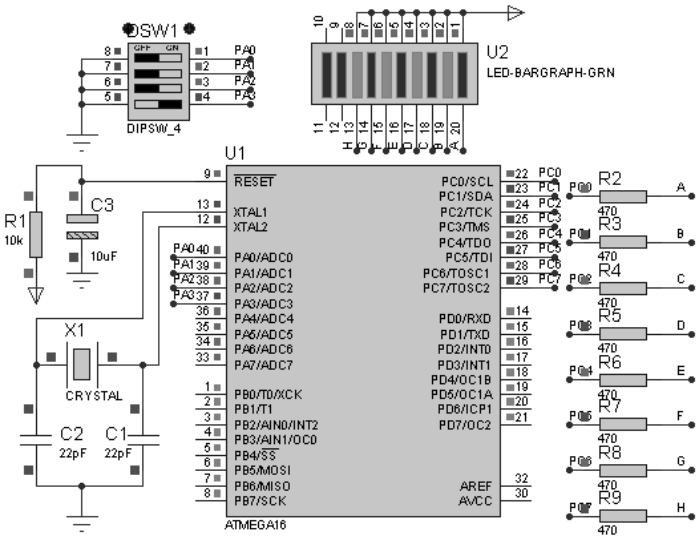


图 5-20　条形 LED 花样选择显示仿真效果图

任务 6　查询式按键设计

将 8 个按键按 1～8 进行编号，如果其中一个键被按下，则在 LED 数码管上显示相应的键值。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 5-6 所列的元器件，并完成图 5-21 所示的硬件电路图设计。

表 5-6　查询式按键设计所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	晶振 CRYSTAL 8MHz	电解电容 CAP-ELEC
电阻 RES	按钮 BUTTON	数码管 7SEG-COM-AN-GRN	

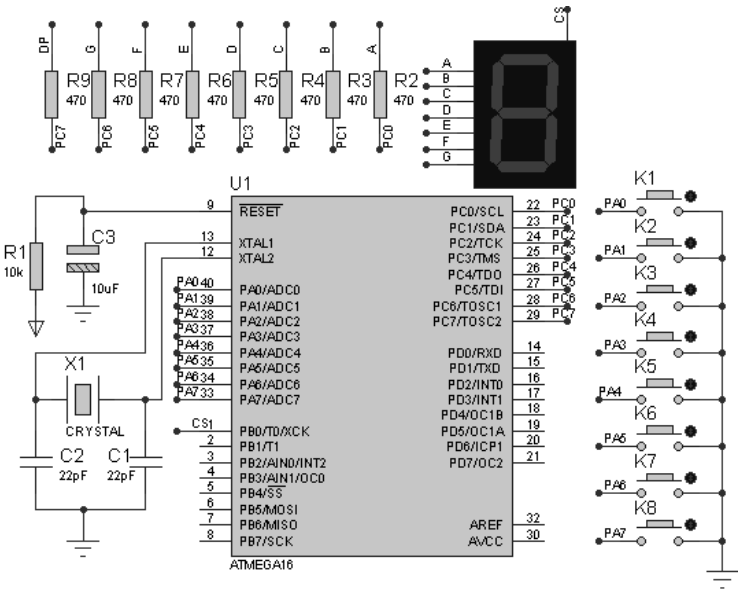


图 5-21　查询式按键电路

软件设计

如果有键被按下，则相应输入为低电平，否则为高电平。这样可通过读入 PINA 口的数据来判断被按下的是什么键。在有键被按下后，要有一定的延时，防止由于键盘抖动而引起的误操作。其程序流程图如图 5-22 所示。

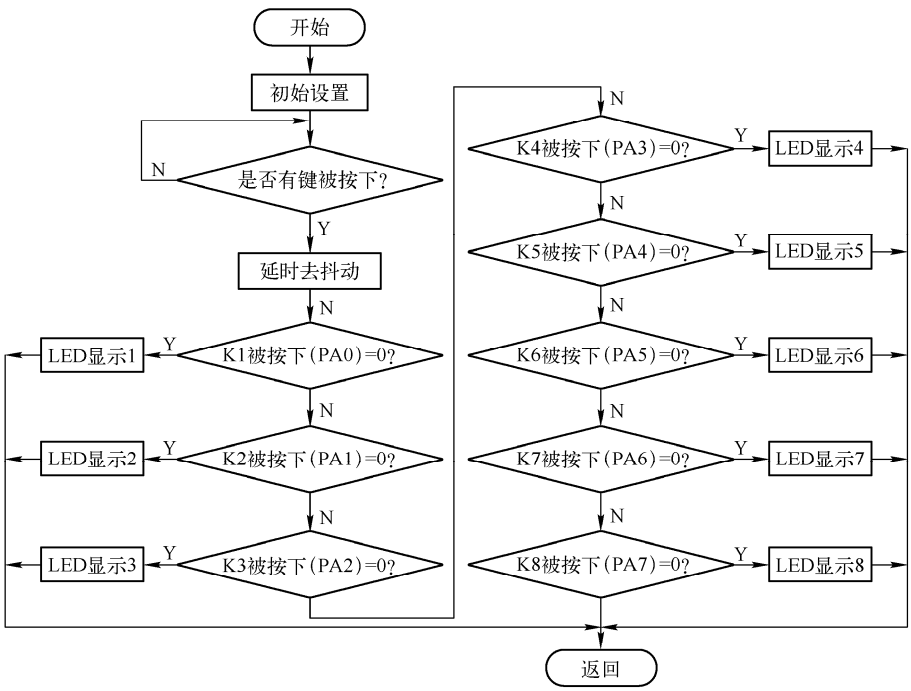


图 5-22　查询式按键程序流程图

PROGRAM 源程序

```
/*****************************************************
 文件名:          查询式按键.c
 单片机型号:      ATmega16
 晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
flash uchar tab[]={
        0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,          //共阳极 LED0～F 的段码
        0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E,0xBF};    //"0xBF"表示"-"
void delay(uint k)                                        //约 1ms 延时函数
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void port_init()                                          //端口初始化
{
  PORTA=0xFF;                                             //PA 端口初始值设置
  DDRA=0x00;                                              //PA 端口配置为输入
  PORTB=0xFE;                                             //PB 端口初始值设置
  DDRB=0xFF;                                              //PB 端口配置为输出
  PORTC=0xFF;                                             //PC 端口初始值设置
  DDRC=0xFF;                                              //PC 端口配置为输出
}
void main(void)
 {
   uchar key;
   port_init();
   PORTB=0xFF;
   while(1)
     {
       while(PINA==0xFF);                                 //等待键被按下
       {
           delay(10);                                     //延时去抖动
        while(PINA==0xFF);
            {
            key=PINA;                                     //读取键值
            switch(key)
             {
                case 0xFE :     PORTC=tab[1];   break;
                case 0xFD :     PORTC=tab[2];   break;
```

```
                case 0xFB :     PORTC=tab[3];   break;
                case 0xF7 :     PORTC=tab[4];   break;
                case 0xEF :     PORTC=tab[5];   break;
                case 0xDF :     PORTC=tab[6];   break;
                case 0xBF :     PORTC=tab[7];   break;
                case 0x7F :     PORTC=tab[8];   break;
            }
        }
    }
}
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“查询式按键.COFF”文件，然后在 Proteus ISIS 中打开已创建的查询式按键电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。此时按下某个按键后，LED 数码管将会显示相应的按键值，如按下按键 K5 后，其运行仿真效果如图 5-23 所示。

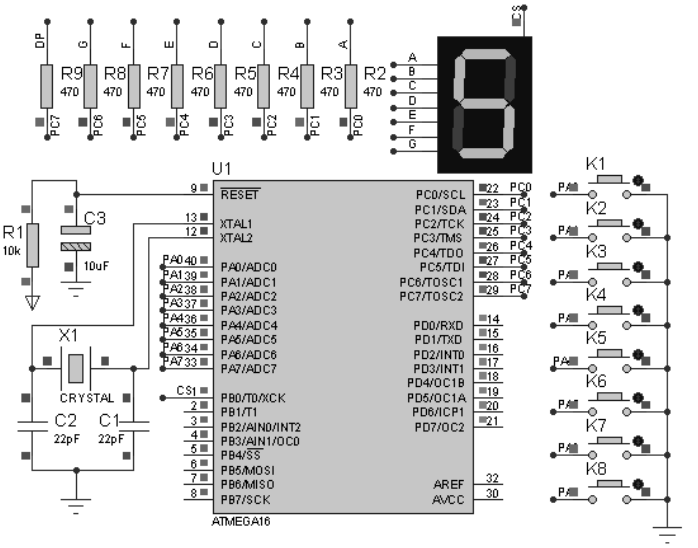


图 5-23　查询式按键运行仿真效果图

任务 7　LED 数码管显示矩阵键盘按键的设计

设计一个 4×4 的矩阵键盘，以 PA0～PA3 作为行线，以 PA4～PA7 作为列线。要求：未按下按键时，LED 数码管显示“-”；按下按键时，在数码管上显示相应的键值。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加

表 5-7 所列的元器件，并完成图 5-24 所示的硬件电路图设计。

表 5-7　LED 数码管显示矩阵键盘按键的所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	晶振 CRYSTAL 8MHz	电解电容 CAP-ELEC
电阻 RES	按钮 BUTTON	数码管 7SEG-COM-AN-GRN	

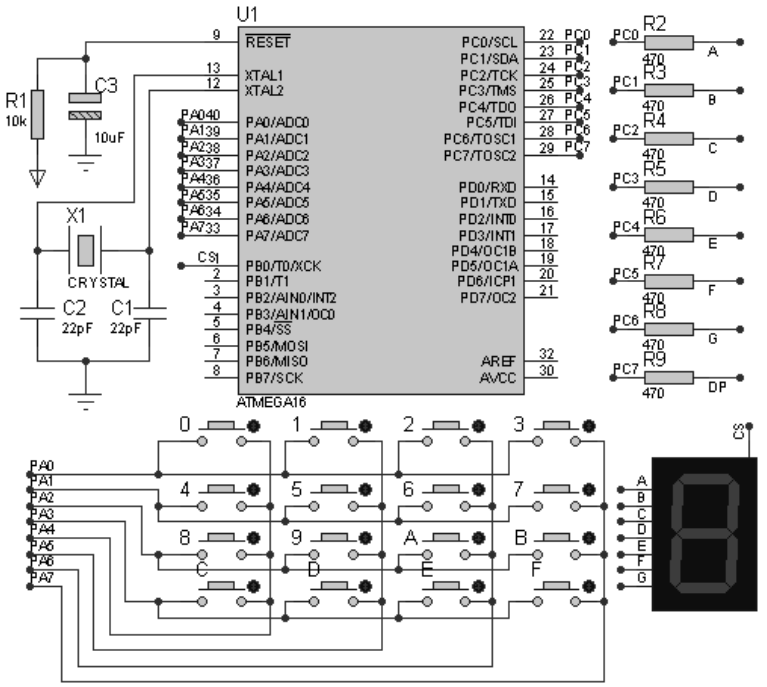


图 5-24　LED 数码管显示矩阵键盘按键电路

以 PA0～PA3 为行线，PA4～PA7 为列线时，可构成 4×4 键盘，即可连接 16 个按键，如图 5-24 所示。行线连接的接口为输入口，用于输入按键的行位置信息；列线连接的接口为输出口，用于输出扫描电平。

矩阵式结构的键盘显然比直接法要复杂一些，识别也要复杂一些。确定矩阵式键盘上何键被按下通常采用行扫描法，行扫描法行扫描法又称为逐行（或列）扫描查询法。



软件设计

如果键盘中的按键数量较多时，为了减少对 I/O 端口的占用，通常将按键排列成矩阵形式。在矩阵式键盘中，每条水平线和垂直线在交叉处不直接连通，而是通过一个按键加以连接。这样，一个端口（如 PA 端口）就可以构成 4×4=16 个按键，比直接将端口线与按键连接的方式多出了一倍，而且线数越多，区别越明显。例如，再多加一条线就可以构成 20 键的键盘，而直接用端口线则只能多出一键（9 键）。由此可见，在需要的键数比较多时，采用矩阵法来设计键盘是合理的。判断键盘中有无键被按下时，先判断到底是哪根行线的状态为低电平，即某根行线为低电平时，其他线为高电平。在确认某行线为低电平后，再逐行检测各列线的电平状态。若某列为低，则该列线与低电平的行线交叉处的按键就是闭合的按

键。LED 数码管显示矩阵键盘按键属于矩阵键盘的应用，核心为键盘扫描。其程序流程图如图 5-25 所示。

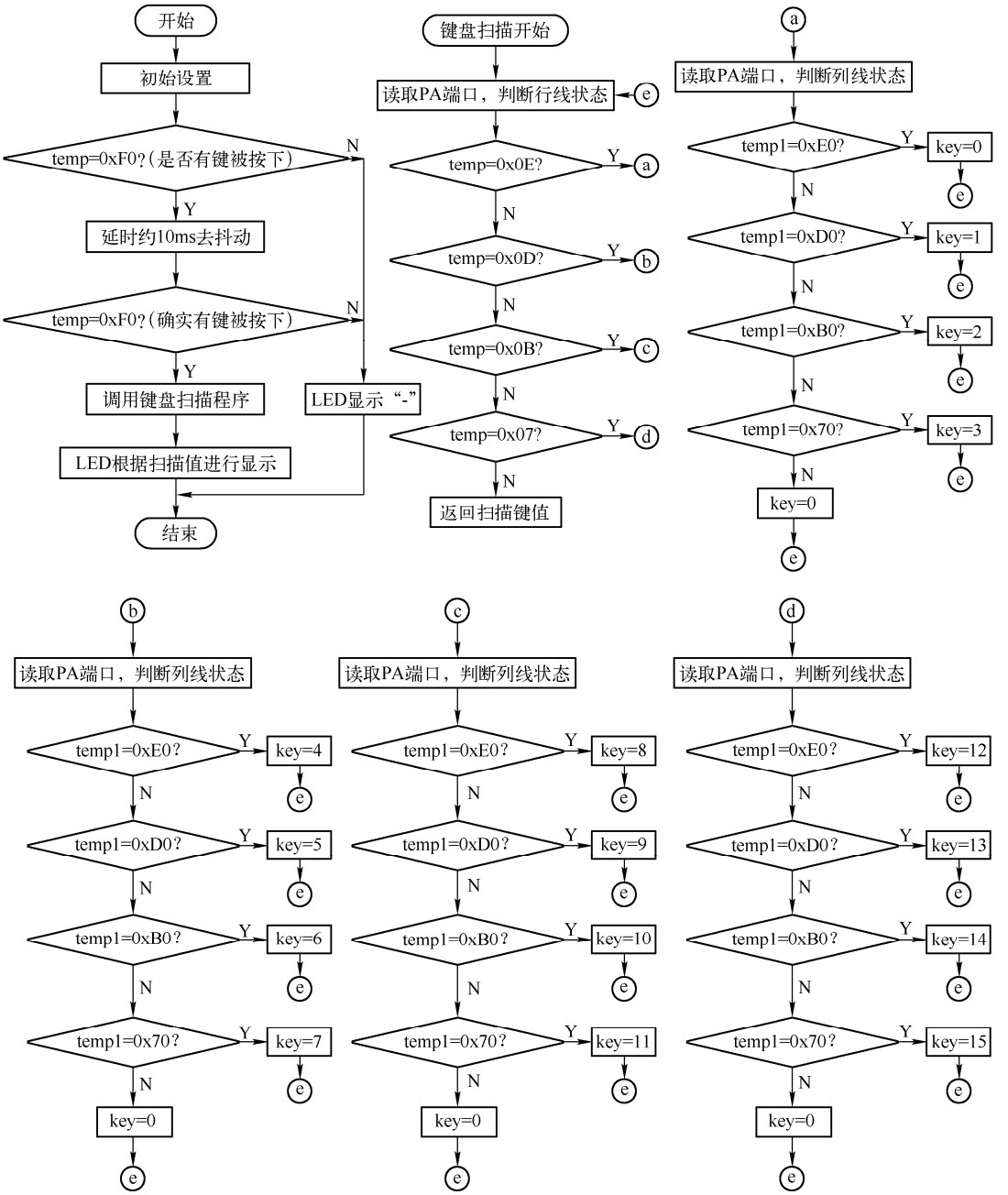


图 5-25　LED 数码管显示矩阵键盘按键程序流程图

PROGRAM 源程序

```
/*****************************************************
文件名:          LED 数码管显示矩阵键盘按键.c
```

```
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
**************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
flash uchar tab[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,      //共阳极 LED0～F 的段码
                   0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E,0xBF,0x10};
uchar dis_buff[1];
bit key_flag;
void    delay(uint k)                          //延时约 0.1ms
{
    uint    m,n;
        for(m=0;m<k;m++)
            {
              for(n=0;n<114;n++);
            }
}
uchar ScanKey()                                //键盘扫描
{
  uchar temp,temp1,key;
  temp=PINA;                                   //读取 PA 端口状态
  temp&=0x0F;                                  //判断行线状态
  switch(temp)
  {
    case 0x0E:                                 //行线的 PA0 为低电平
      {
          DDRA=0x0F;                           //改变端口方向，准备进行列线的状态判断
         PORTA=0xF0;
          delay(5);                            //延时片刻
         temp1=PINA;                           //读取 PA 端口状态
        temp1&=PINA;                           //判断列线状态
        switch(temp1)
          {
            case 0xE0:                         //列线的 PA4 为低电平，表示“0”键被按下
                key=0;                         //键值为 0
                break;                         //跳出列线的状态判断
            case 0xD0:                         //列线的 PA5 为低电平，表示“1”键被按下
                key=1;                         //键值为 1
                break;                         //跳出列线的状态判断
            case 0xB0:                         //列线的 PA6 为低电平，表示“2”键被按下
                key=2;
                break;
            case 0x70:                         //列线的 PA7 为低电平，表示“3”键被按下
                key=3;
                break;
           default:                            //列线没发生状态改变
                key=0;
```

```
                break;
            }
            DDRA=0xF0;                          //改变 PA 端口方向，为下次行线判断做好准备
            PORTA=0x0F;
            break;
        }
    case 0x0D:                                  //行线的 PA1 为低电平
        {
            DDRA=0x0F;                          //改变端口方向，准备进行列线的状态判断
            PORTA=0xF0;
            delay(5);
            temp1=PINA;                         //读取 PA 端口状态
           temp1&=PINA;                         //判断列线状态
           switch(temp1)
              {
               case 0xE0:                       //列线的 PA4 为低电平，表示“4”键被按下
                 key=4;
                 break;
               case 0xD0:                       //列线的 PA5 为低电平，表示“5”键被按下
                 key=5;
                 break;
               case 0xB0:                       //列线的 PA6 为低电平，表示“6”键被按下
                 key=6;
                 break;
               case 0x70:                       //列线的 PA7 为低电平，表示“7”键被按下
                 key=7;
                 break;
              default:                          //列线没发生状态改变
                 key=0;
                 break;
              }
            DDRA=0xF0;                          //改变 PA 端口方向，为下次行线判断做好准备
            PORTA=0x0F;
            break;
        }
    case 0x0B:                                  //行线的 PA2 为低电平
        {
            DDRA=0x0F;                          //改变端口方向，准备进行列线的状态判断
            PORTA=0xF0;
            delay(5);
            temp1=PINA;                         //读取 PA 端口状态
           temp1&=PINA;                         //判断列线状态
           switch(temp1)
              {
               case 0xE0:                       //列线的 PA4 为低电平，表示“8”键被按下
                 key=8;
                 break;
               case 0xD0:                       //列线的 PA5 为低电平，表示“9”键被按下
                 key=9;
```

```
                break;
               case 0xB0:                    //列线的 PA6 为低电平，表示“A”键被按下
                key=10;
                break;
               case 0x70:                    //列线的 PA7 为低电平，表示“B”键被按下
                key=11;
                break;
              default:                       //列线没发生状态改变
                key=0;
                break;
              }
             DDRA=0xF0;                     //改变 PA 端口方向，为下次行线判断做好准备
             PORTA=0x0F;
             break;
        }
      case 0x07:                            //行线的 PA3 为低电平
        {
             DDRA=0x0F;                     //改变端口方向，准备进行列线的状态判断
            PORTA=0xF0;
             delay(5);
            temp1=PINA;                     //读取 PA 端口状态
           temp1&=PINA;                     //判断列线状态
           switch(temp1)
             {
               case 0xE0:                   //列线的 PA4 为低电平，表示“C”键被按下
                key=12;
                break;
               case 0xD0:                   //列线的 PA5 为低电平，表示“D”键被按下
                key=13;
                break;
               case 0xB0:                   //列线的 PA6 为低电平，表示“E”键被按下
                key=14;
                break;
               case 0x70:                   //列线的 PA7 为低电平，表示“F”键被按下
                key=15;
                break;
               default:                     //列线没发生状态改变
                key=0;
                break;
              }
             DDRA=0xF0;                     //改变 PA 端口方向，为下次行线判断做好准备
             PORTA=0x0F;
             break;
        }
       default:                             //行线没发生状态改变
       key=0;
       break;
  }
  return(key);                              //返回键值
}
```

```
void port_init()                          //端口初始化
{
  PORTA=0x0F;                             //PA 端口初始值设置
  DDRA=0xF0;                              //PA 端口配置
  PORTB=0xFE;                             //PB 端口初始值设置
  DDRB=0xFF;                              //PB 端口配置为输出
  PORTC=0xFF;                             //PC 端口初始值设置
  DDRC=0xFF;                              //PC 端口配置为输出
}
void display(void)
{
  PORTB=0x01;
  PORTC=tab[dis_buff[0]];                 //根据键值进行 LED 数码管显示
  delay(10);
}
void main(void)
{
  uchar temp,keynum;
  port_init();
  while (1)
    {
      temp=PINA;                          //判断是否有按键
      temp&=0x0F;
      if(temp==0x0F)
        {
          key_flag=0;
        }
      else
        {
          delay(100);                     //约 10ms 延时去抖
          temp=PINA;                      //判断是否有按键被按下
          temp&=0x0F;
        }
      if(temp==0x0F)
        {
          key_flag=0;
        }
      else
        {
          keynum=ScanKey();               //键值送 keynum
          key_flag=1;
          PORTC=tab[keynum];              //根据键值进行 LED 数码管显示
        }
      while(temp!=0x0F)
        {
          temp=PINA;
          temp&=0x0F;
        }
      if(key_flag==0)
        {
```

```
            dis_buff[0]=0x10;
        }
      if(key_flag==1)
        {
          dis_buff[0]=keynum;                    //显示新按下键的键值
        }
      display();
    }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“LED 数码管显示矩阵键盘按键.COFF”文件，然后在 Proteus ISIS 中打开已创建的 LED 数码管显示矩阵键盘按键电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。此时，没有按下按键时，LED 数码管显示“-”；当按下某个按键时，LED 数码管将显示相应的按键值，如按下按键 6 后，其运行仿真效果如图 5-26 所示；当松开按键后，LED 数码管仍显示“-”。

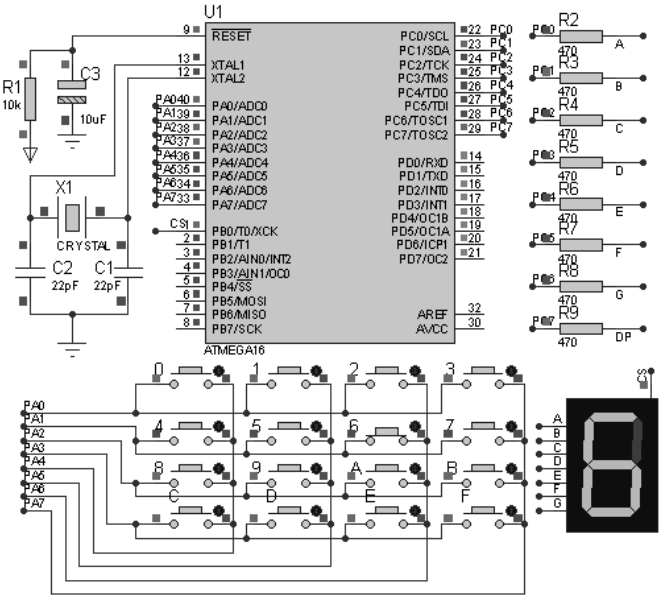


图 5-26　LED 数码管显示矩阵键盘按键的运行仿真效果图

任务 8　矩阵键盘按键控制 LED+条形 LED 的设计

设计一个矩阵键盘按键控制 LED+条形 LED 电路，PA 端口作为 4×4 矩阵键盘接口，PD0～PD3 控制 16 位的 LED+条形 LED 的亮与灭，PC 端口控制 LED 数码管显示。其中，PD0～PD3 作为行线，PD4～PD7 作为列线。要求：初始状态下，LED 数码管显示 0，条形 LED 第 0 位亮，其余不亮；按下某个按键后，不管按键是否松开，LED 数码管均显示该键值，且条形 LED 相应位亮，而其余位不亮。

硬件设计

PD0～PD3 只有 4 个 I/O 端口位，要控制 16 位的 LED+条形 LED 的亮与灭，则需使用 4-16 线的译码器，在此可选用 74154 芯片。

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 5-8 所列的元器件，并完成图 5-27 所示的硬件电路图设计。

表 5-8　矩阵键盘按键控制 LED+条形 LED 所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	晶振 CRYSTAL 8MHz	电解电容 CAP-ELEC
电阻 RES	按钮 BUTTON	数码管 7SEG-COM-AN-GRN	4-16 线译码器 74154
条形 LED LED-BARGRAPH-GRN			

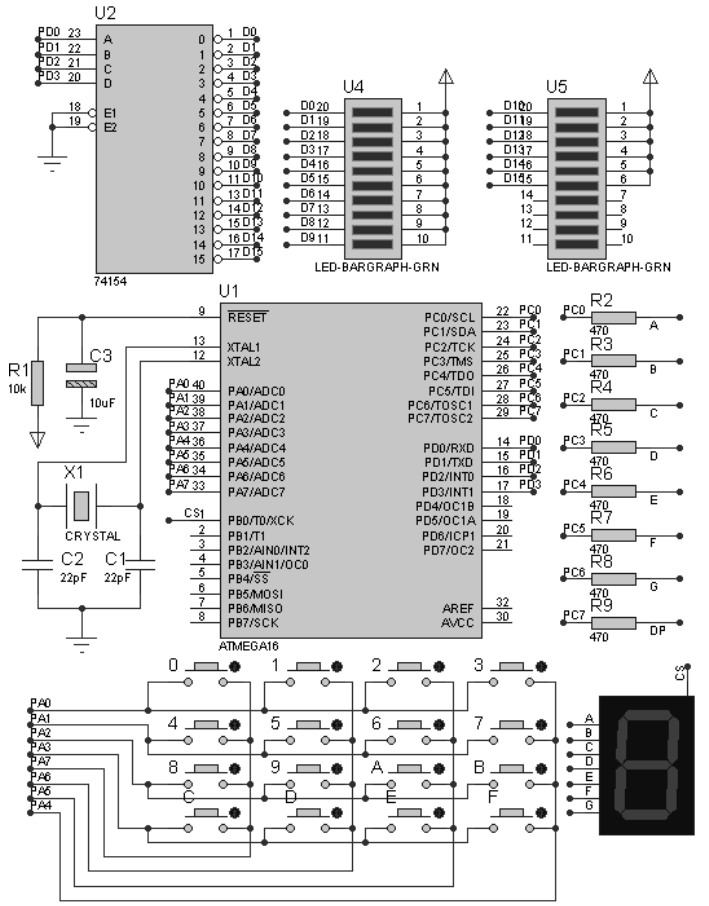


图 5-27　矩阵键盘按键控制 LED+条形 LED 电路

软件设计

矩阵键盘按键控制 LED+条形 LED 实质上是 4×4 矩阵键盘的应用，其核心应该也属于键盘扫描，也是先判断某一行的状态为低电平，再判断某一列的状态也为低电平时，则该列线与低电平线的行线交叉处的按键就是闭合的按键。其程序流程图请参考图 5-25。

PROGRAM 源程序

```
/****************************************************
 文件名:          矩阵键盘按键控制 LED+条形 LED.c
 单片机型号:      ATmega16
 晶振频率:        8.0MHz
****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
flash uchar tab[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,  //共阳极 LED0～F 的段码
                   0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E,0xBF,0x10};
flash uchar tab1[]={0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,
                    0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F};
uchar dis_buff[1];
void   delay(uint k)                          //延时约 0.1ms
{
    uint   m,n;
        for(m=0;m<k;m++)
            {
              for(n=0;n<114;n++);
            }
}
uchar ScanKey()                               //键盘扫描
{
  uchar temp,temp1,key;
  temp=PINA;                                  //读取 PA 端口状态
  temp&=0x0F;                                 //判断行线状态
  switch(temp)
  {
    case 0x0E:                                //行线的 PA0 为低电平
      {
         DDRA=0x0F;                           //改变端口方向，准备进行列线的状态判断
        PORTA=0xF0;
         delay(5);                            //延时片刻
        temp1=PINA;                           //读取 PA 端口状态
       temp1&=PINA;                           //判断列线状态
       switch(temp1)
         {
          case 0x70:                          //列线的 PA7 为低电平，表示“0”键被按下
              key=0;                          //键值为 0
              break;                          //跳出列线的状态判断
          case 0xB0:                          //列线的 PA6 为低电平，表示“1”键被按下
              key=1;                          //键值为 1
              break;                          //跳出列线的状态判断
          case 0xD0:                          //列线的 PA5 为低电平，表示“2”键被按下
              key=2;
              break;
```

```
            case 0xE0:                    //列线的 PA4 为低电平，表示“3”键被按下
                key=3;
                break;
            default:                      //列线没发生状态改变
                key=0;
                break;
            }
          DDRA=0xF0;                      //改变 PA 端口方向，为下次行线判断做好准备
          PORTA=0x0F;
          break;
    }
case 0x0D:                                //行线的 PA1 为低电平
    {
          DDRA=0x0F;                      //改变端口方向，准备进行列线的状态判断
          PORTA=0xF0;
          delay(5);
          temp1=PINA;                     //读取 PA 端口状态
        temp1&=PINA;                      //判断列线状态
        switch(temp1)
          {
            case 0x70:                    //列线的 PA7 为低电平，表示“4”键被按下
              key=4;
              break;
            case 0xB0:                    //列线的 PA6 为低电平，表示“5”键被按下
              key=5;
              break;
            case 0xD0:                    //列线的 PA5 为低电平，表示“6”键被按下
              key=6;
              break;
            case 0xE0:                    //列线的 PA4 为低电平，表示“7”键被按下
              key=7;
              break;
            default:                      //列线没发生状态改变
              key=0;
              break;
            }
          DDRA=0xF0;                      //改变 PA 端口方向，为下次行线判断做好准备
          PORTA=0x0F;
          break;
    }
case 0x0B:                                //行线的 PA2 为低电平
    {
          DDRA=0x0F;                      //改变端口方向，准备进行列线的状态判断
          PORTA=0xF0;
          delay(5);
          temp1=PINA;                     //读取 PA 端口状态
        temp1&=PINA;                      //判断列线状态
        switch(temp1)
```

```
        {
         case 0x70:                    //列线的 PA7 为低电平，表示“8”键被按下
          key=8;
          break;
         case 0xB0:                    //列线的 PA6 为低电平，表示“9”键被按下
          key=9;
          break;
         case 0xD0:                    //列线的 PA5 为低电平，表示“A”键被按下
          key=10;
          break;
         case 0xE0:                    //列线的 PA4 为低电平，表示“B”键被按下
          key=11;
          break;
        default:                       //列线没发生状态改变
          key=0;
          break;
        }
       DDRA=0xF0;                      //改变 PA 端口方向，为下次行线判断做好准备
       PORTA=0x0F;
       break;
  }
case 0x07:                             //行线的 PA3 为低电平
  {
       DDRA=0x0F;                      //改变端口方向，准备进行列线的状态判断
      PORTA=0xF0;
       delay(5);
      temp1=PINA;                      //读取 PA 端口状态
     temp1&=PINA;                      //判断列线状态
     switch(temp1)
       {
        case 0x70:                     //列线的 PA7 为低电平，表示“C”键被按下
          key=12;
          break;
         case 0xB0:                    //列线的 PA6 为低电平，表示“D”键被按下
          key=13;
          break;
         case 0xD0:                    //列线的 PA5 为低电平，表示“E”键被按下
          key=14;
          break;
         case 0xE0:                    //列线的 PA4 为低电平，表示“F”键被按下
          key=15;
          break;
         default:                      //列线没发生状态改变
          key=0;
          break;
       }
      DDRA=0xF0;                       //改变 PA 端口方向，为下次行线判断做好准备
      PORTA=0x0F;
      break;
  }
```

```
        default:                          //行线没发生状态改变
        key=0;
        break;
    }
    return(key);                          //返回键值
}
void port_init()                          //端口初始化
{
    PORTA=0x0F;                           //PA 端口初始值设置
    DDRA=0xF0;                            //PA 端口配置
    PORTB=0xFE;                           //PB 端口初始值设置
    DDRB=0xFF;                            //PB 端口配置为输出
    PORTC=0xFF;                           //PC 端口初始值设置
    DDRC=0xFF;                            //PC 端口配置为输出
    PORTD=0xFF;                           //PD 端口初始值设置
    DDRD=0xFF;                            //PD 端口配置为输出
}
void display(void)
{
    PORTB=0x01;
    PORTC=tab[dis_buff[0]];               //根据键值进行 LED 数码管显示
    PORTD=tab1[dis_buff[0]];
    PORTD=dis_buff[0];
    delay(10);
}
void main(void)
{
    uchar temp,keynum;
    port_init();
    while (1)
      {
        temp=PINA;                        //判断是否有按键被按下
        temp&=0x0F;
        if(temp==0x0F)
          {
            continue;
          }
        else
          {
            delay(100);                   //约 10ms 延时去抖
            temp=PINA;                    //判断是否有按键
            temp&=0x0F;
          }
        if(temp==0x0F)
          {
            continue;
          }
        else
          {
            keynum=ScanKey();             //键值送 keynum
```

```
            dis_buff[0]=keynum;                    //根据键值进行 LED 数码管显示
        }
      while(temp!=0x0F)
        {
          temp=PINA;
          temp&=0x0F;
        }
      display();
    }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“矩阵键盘按键控制 LED+条形 LED.COFF”文件，然后在 Proteus ISIS 中打开已创建的矩阵键盘按键控制 LED+条形 LED 电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。此时，没有按下按键时，LED 数码管显示“0”，条形 LED 第 16 位点亮；当按了某个按键后，LED 数码管将显示相应的按键值，且条形 LED 相应位点亮，如按下按键 9 后，其运行仿真效果如图 5-28 所示。

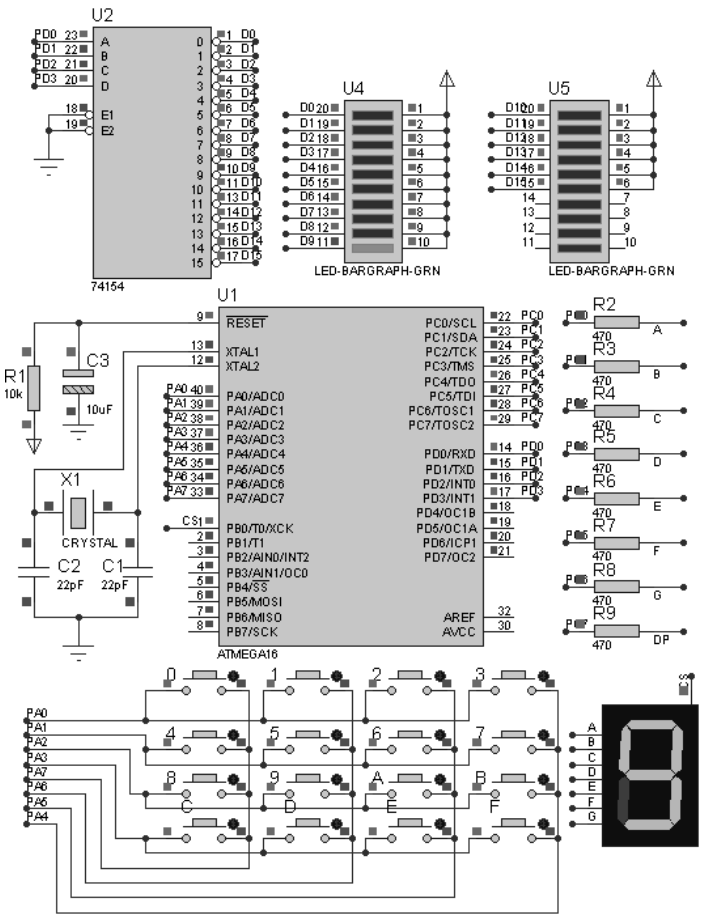


图 5-28　矩阵键盘按键控制 LED 条形 LED 仿真效果图

任务 9　简单拨号键盘的设计

使用 4×4 矩阵式键盘及 8 位共阳极 LED 数码管设计一个简单拨号键盘，要求在初始状态时（系统刚上电），8 位 LED 数码管均显示“-”，每按下一个号码后，原 8 位 LED 数码管的显示内容向左移动 1 位。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 5-9 所列的元器件，并完成图 5-29 所示的硬件电路图设计。

表 5-9　简单拨号键盘所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	晶振 CRYSTAL 8MHz	电解电容 CAP-ELEC
电阻 RES	三极管 NPN	数码管 7SEG-MPX8-CA-BLUE	按钮 BUTTON

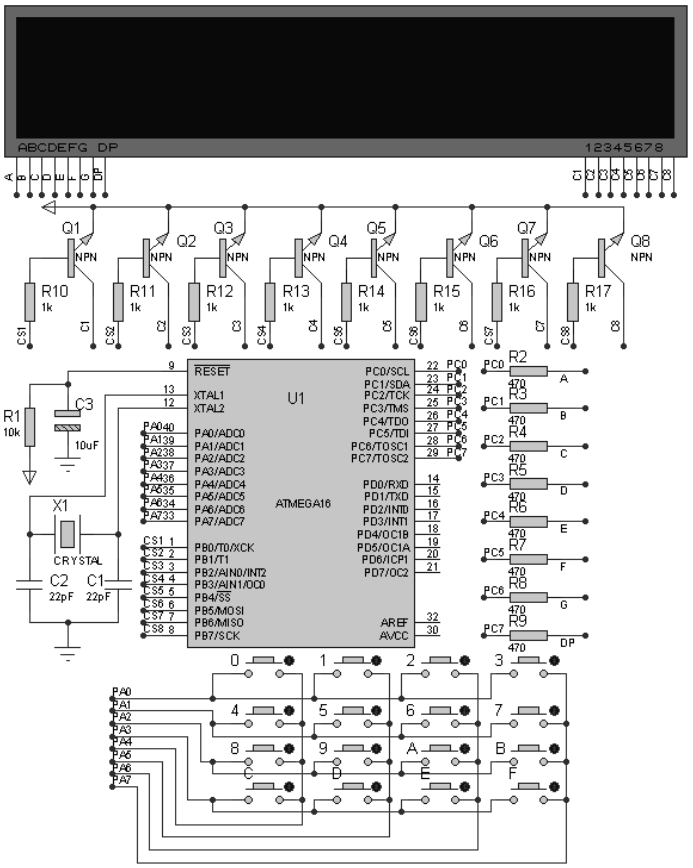


图 5-29　简单拨号键盘电路

软件设计

该设计主要由键盘扫描、LED 扫描显示子程序和主程序构成。键盘扫描程序可以参照

任务 7 中的源代码。LED 扫描显示程序，可按顺序先后对 PORTB 进行 LED 片选设置，同时将显示数据送到 PORTC，并延时 1ms 即可。

在主程序中，先进行相应的初始化，然后再判断是否有键被按下。初始化过程主要是 8 位 LED 显示初值设置。LED 的显示初值为“-”，因此使用循环语句执行“dis_buff[x]=16”8 次，使 8 位 LED 均显示“-”。

在判断是否有键被按下时，如果没有键被按下，则 8 位 LED 的显示状态保持前一时刻的状态；如果有键被按下时，8 位 LED 的显示内容左移 1 位，其中新的按键值送给 dis_buff[7]即可。主程序的程序流程图如图 5-30 所示，键盘扫描部分的流程图请参照图 5-25。

注意:

由于图 5-29 中键盘的列线发生了改变，因此在编写键盘扫描程序代码时，其判断值也应该发生相应改变。

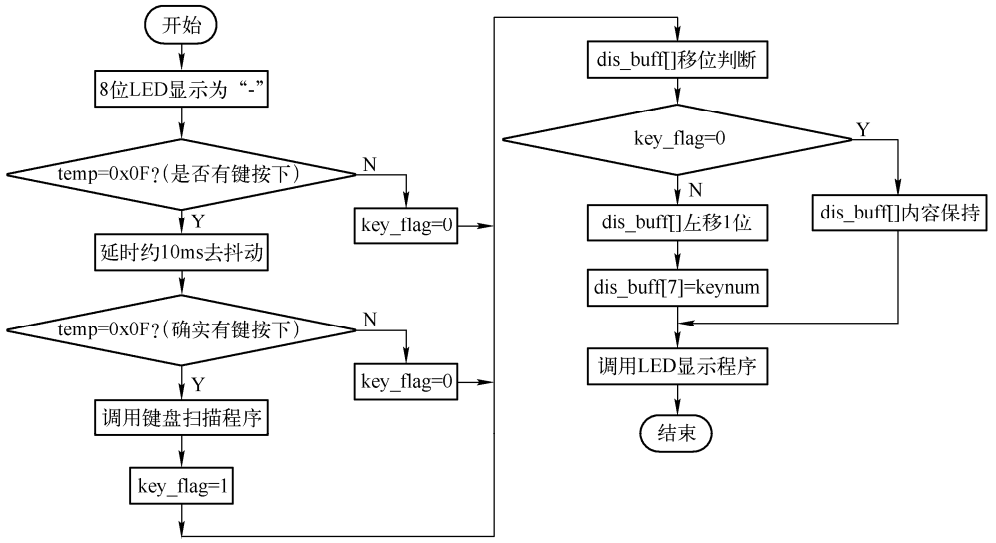


图 5-30　简单拨号键盘的设计主程序流程图

PROGRAM 源程序

```
/*****************************************************
  文件名:          简单拨号键盘.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
flash uchar tab[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,      //共阳极 LED   0～F 的段码
                   0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E,0xBF,};
uchar dis_buff[8];
bit key_flag;
void   delay(uint k)                          //延时约 0.1ms
```

```
{
    uint   m,n;
        for(m=0;m<k;m++)
            {
              for(n=0;n<114;n++);
            }
}
void display(void)
{
  uchar i,j;
  j=0x01;
  for(i=0;i<8;i++)
    {
      PORTB=0x00;
      PORTC=tab[dis_buff[i]];
      PORTB=j;
      j=j<<1;
      delay(1);
    }
}
uchar ScanKey()                          //键盘扫描
{
  uchar temp,temp1,key;
  temp=PINA;                             //读取 PA 端口状态
  temp&=0x0F;                            //判断行线状态
  switch(temp)
  {
    case 0x0E:                           //行线的 PA0 为低电平
      {
         DDRA=0x0F;                      //改变端口方向，准备进行列线的状态判断
        PORTA=0xF0;
         delay(5);                       //延时片刻，以利于状态的获取
        temp1=PINA;                      //读取 PA 端口状态
       temp1&=PINA;                      //判断列线状态
       switch(temp1)
         {
          case 0x70:                     //列线的 PA7 为低电平，表示“0”键被按下
              key=0;                     //键值为 0
              break;                     //跳出列线状态判断
          case 0xB0:                     //列线的 PA6 为低电平，表示“1”键被按下
              key=1;                     //键值为 1
              break;                     //跳出列线状态判断
          case 0xD0:                     //列线的 PA5 为低电平，表示“2”键被按下
              key=2;
              break;
          case 0xE0:                     //列线的 PA4 为低电平，表示“3”键被按下
              key=3;
              break;
```

```
        default:                    //列线没发生状态改变
            key=0;
            break;
        }
      DDRA=0xF0;                    //改变 PA 端口方向，为下次行线判断做好准备
      PORTA=0x0F;
      break;
  }
case 0x0D:                          //行线的 PA1 为低电平
  {
      DDRA=0x0F;                    //改变端口方向，准备进行列线的状态判断
     PORTA=0xF0;
      delay(5);
    temp1=PINA;                     //读取 PD 端口状态
   temp1&=PINA;                     //判断列线状态
   switch(temp1)
     {
      case 0x70:                    //列线的 PA7 为低电平，表示“4”键被按下
        key=4;
        break;
      case 0xB0:                    //列线的 PA6 为低电平，表示“5”键被按下
        key=5;
        break;
      case 0xD0:                    //列线的 PA5 为低电平，表示“6”键被按下
        key=6;
        break;
      case 0xE0:                    //列线的 PA4 为低电平，表示“7”键被按下
        key=7;
        break;
      default:                      //列线没发生状态改变
        key=0;
        break;
     }
      DDRA=0xF0;                    //改变 PA 端口方向，为下次行线判断做好准备
      PORTA=0x0F;
      break;
  }
case 0x0B:                          //行线的 PA2 为低电平
  {
      DDRA=0x0F;                    //改变端口方向，准备进行列线的状态判断
     PORTA=0xF0;
      delay(5);
    temp1=PINA;                     //读取 PD 端口状态
   temp1&=PINA;                     //判断列线状态
   switch(temp1)
     {
      case 0x70:                    //列线的 PA7 为低电平，表示“8”键被按下
        key=8;
```

```
            break;
          case 0xB0:                    //列线的 PA6 为低电平，表示“9”键被按下
            key=9;
            break;
          case 0xD0:                    //列线的 PA5 为低电平，表示“A”键被按下
            key=10;
            break;
          case 0xE0:                    //列线的 PA4 为低电平，表示“B”键被按下
            key=11;
            break;
         default:                       //列线没发生状态改变
            key=0;
            break;
          }
        DDRA=0xF0;                      //改变 PA 端口方向，为下次行线判断做好准备
        PORTA=0x0F;
        break;
    }
case 0x07:                              //行线的 PA3 为低电平
    {
         DDRA=0x0F;                     //改变端口方向，准备进行列线的状态判断
        PORTA=0xF0;
         delay(5);
        temp1=PINA;                     //读取 PA 端口状态
       temp1&=PINA;                     //判断列线状态
       switch(temp1)
          {
           case 0x70:                   //列线的 PA7 为低电平，表示“C”键被按下
            key=12;
            break;
           case 0xB0:                   //列线的 PA6 为低电平，表示“D”键被按下
            key=13;
            break;
           case 0xD0:                   //列线的 PA5 为低电平，表示“E”键被按下
            key=14;
            break;
           case 0xE0:                   //列线的 PA4 为低电平，表示“F”键被按下
            key=15;
            break;
           default:                     //列线没发生状态改变
            key=0;
            break;
          }
        DDRA=0xF0;                      //改变 PA 端口方向，为下次行线判断做好准备
        PORTA=0x0F;
        break;
    }
    default:                            //行线没发生状态改变
    key=0;
    break;
```

```
    }
    return(key);                          //返回键值
}
void port_init()                          //端口初始化
{
    PORTA=0x0F;                           //PA 端口初始值设置
    DDRA=0xF0;                            //PA 端口配置
    PORTB=0xFE;                           //PB 端口初始值设置
    DDRB=0xFF;                            //PB 端口配置为输出
    PORTC=0xFF;                           //PC 端口初始值设置
    DDRC=0xFF;                            //PC 端口配置为输出
}
void main(void)
{
    uchar x;
    uchar temp,keynum;
    port_init();
     for(x=0;x<8;x++)                     //初始状态显示 8 个"-"
       {
        dis_buff[x]=16;
       }
    while (1)
       {
         temp=PINA;                       //判断是否有按键
         temp&=0x0F;
         if(temp==0x0F)
           {
              key_flag=0;
           }
         else
           {
              delay(100);                 //约 10ms 延时去抖
              temp=PINA;                  //判断是否有按键
              temp&=0x0F;}
         if(temp==0x0F)
           {
              key_flag=0;
           }
         else
           {
              key_flag=1;
              keynum=ScanKey();           //键值送 keynum
           }
         while(temp!=0x0F)
            {
              temp=PINA;
              temp&=0x0F;
            }
         if(key_flag==1)
            {
```

```
                dis_buff[0]=dis_buff[1];
                dis_buff[1]=dis_buff[2];
                dis_buff[2]=dis_buff[3];
                dis_buff[3]=dis_buff[4];
                dis_buff[4]=dis_buff[5];
                dis_buff[5]=dis_buff[6];
                dis_buff[6]=dis_buff[7];
                dis_buff[7]=keynum;
            }
        display();
    }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“简单拨号键盘.COFF”文件，然后在 Proteus ISIS 中打开已创建的简单拨号键盘电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。此时，没有按下按键时，8 位 LED 数码管均显示“-”，当按下某键时，将显示移位相应的数值，仿真效果如图 5-31 所示。

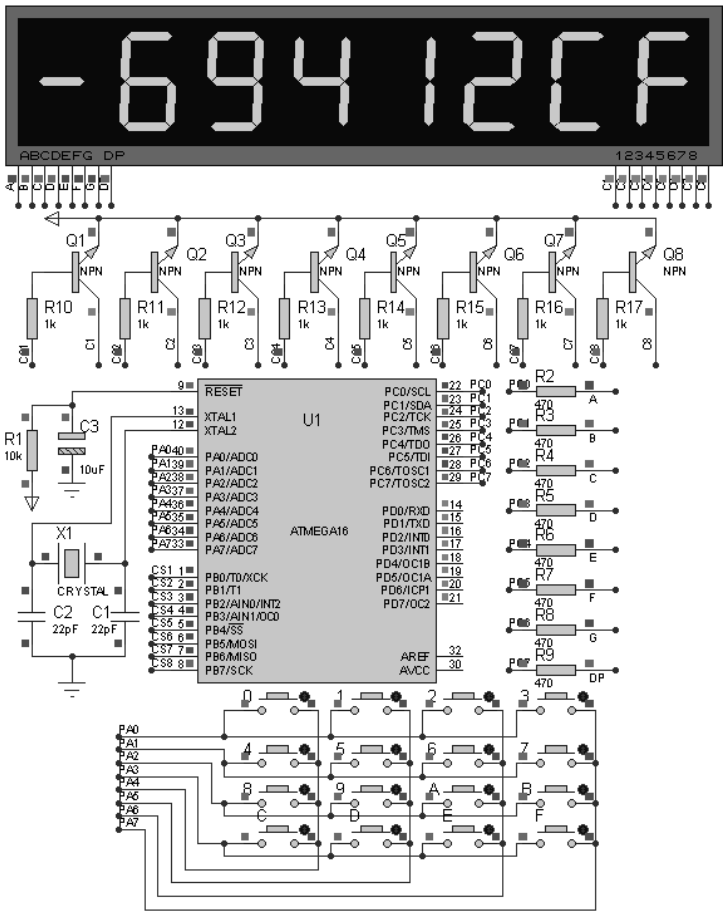


图 5-31　简单拨号键盘仿真效果图

项目六　中断控制的应用设计

〖知识目标〗
☺了解中断定义、中断寄存器定义、中断类型；
☺掌握外部中断控制。
〖能力目标〗
☺学会中断相关寄存器的设置；
☺会利用外部中断实现对单片机的控制。



任务 1　中断控制原理

1．中断、中断源和中断向量

所谓中断，是指当计算机执行正常程序时，系统中出现某些急需处理的异常情况和特殊请求，CPU 暂时中止现行程序，转去对随机发生的更为紧迫的事件进行处理，处理完毕后，CPU 自动返回原来的程序继续执行的过程。

能向 CPU 发出请求的事件称为中断源。计算机系统中一般有多个中断源，ATmega16 单片机可直接处理的有 21 个中断源。每个中断源都有独立的中断向量。默认情况下，AVR 程序存储区的最低端（即从 Flash 地址的 0x0000 开始）用于放置中断向量，称之为中断向量区。各种型号的 AVR 中断向量区的大小是不同的，可使用下式进行计算：

中断向量区大小=中断源个数×每个中断向量占据字数

ATmega16 片内的 Flash 为 8K 字（word），因此每个中断向量占据了 2 字（即 4B）。ATmega16 的中断源及每个中断源所对应的中断向量见表 6-1。

表 6-1　ATmega16 的中断源及中断向量

中断向量号	程序地址[注2]	中　断　源	中断定义说明
1	0x0000[注1]	RESET	外部引脚电平引发的复位、上电复位、看门狗复位及 JTAG AVR 复位
2	0x0002	INT0	外部中断请求 0
3	0x0004	INT1	外部中断请求 1
4	0x0006	TIMER2 COMP	定时器/计数器 2 比较匹配
5	0x0008	TIMER2 OVF	定时器/计数器 2 溢出
6	0x000A	TIMER1 CAPT	定时器/计数器 2 事件捕捉
7	0x000C	TIMER1 COMPA	定时器/计数器 1 比较匹配 A
8	0x000E	TIMER1 COMPB	定时器/计数器 1 比较匹配 B
9	0x0010	TIMER1 OVF	定时器/计数器 1 溢出

续表

中断向量号	程序地址[注2]	中　断　源	中断定义说明
10	0x0012	TIMER0 OVF	定时器/计数器 0 溢出
11	0x0014	SPI STC	SPI 串行传输结束
12	0x0016	USART RXC	USART，接收结束
13	0x0018	USART UDRE	USART，数据寄存器空
14	0x001A	USART TXC	USART，发送结束
15	0x001C	ADC	A/D 转换结束
16	0x001E	EE_RDY	E^2PROM 就绪
17	0x0020	ANA_COMP	模拟比较器
18	0x0022	TWI	两线串行接口
19	0x0024	INT2	外部中断请求 2
20	0x0026	TIMER0 COMP	定时器/计数器 0 比较匹配
21	0x0028	SPM_RDY	保存程序存储器内容就绪

注 1：熔丝位 BOOTRST 被编程时，ATmega16 复位后程序将跳转到 BootLoader。

注 2：当寄存器 GICR 的 IVSEL 置位时，中断向量转移到 Boot 区的起始地址，此时各个中断向量的实际地址为表中地址与 Boot 区起始地址之和，见表 6-2。

表 6-2　中断向量地址的设置

BOOTRST	IVSEL	复位地址	中断向量起始地址
1	0	0x0000	0x0002
1	1	0x0000	Boot 复位地址+0x0002
0	0	Boot 复位地址	0x0002
0	1	Boot 复位地址	Boot 复位地址+0x0002

在 ATmega16 的 21 个中断源中，包含 1 个非屏蔽的系统复位中断（RESET）、3 个外部中断（INT0、INT1 和 INT2）和 17 个内部中断。

系统复位中断 RESET 是一个特殊的中断源，当 ATmega16 由于各种原因被复位后，程序将跳到复位中断地址（默认为 0x0000），在该地址处通常放置一条跳转指令，用于跳转到主程序继续执行。

3 个外部中断（INT0、INT1 和 INT2）由 ATmega16 的外部引脚 PD2、PD3 和 PB2 的电平变化或状态触发。通过设置控制寄存器 MCUCR、控制与状态寄存器 MCUCSR 的相关位，可以定义这些外部中断的触发方式。

TIMER2 COMP，TIMER2 OVF，TIMER1 CAPT，TIMER1 COMPA，TIMER1 COMPB，TIMER1 OVF，TIMER0 OVF，TIMER0 COMP 这 8 个内部中断源是来自于 ATmega16 的内部定时器/计数器（C/T2、C/T1 和 C/T0）触发的内部中断。

USART RXC、USART UDRE、USART TXC 是来自于 ATmega16 内部的通用同步/异步串行接收的转发器 USART 的 3 个内部中断。当 USART 串口完整接收 1B 数据，发送数据寄存器为空，成功发送完 1B 数据时，这 3 个中断会分别被触发。

SPI STC 为内部 SPI 串行接口传送结束中断；ADC 为 ADC 单元完成 A/D 转换的中断；EE_RDY 为片内 E^2PROM 就绪（对 E^2PROM 的操作完成）中断；ANA_COMP 是由内置的模拟比较器输出引发的中断；TWI 为内部两线串行接口的中断；SPM_RDY 是对片内的

Flash 写操作完成的中断。

2．中断处理流程和中断优先级

由于各计算机系统的中断系统硬件结构不同，所以中断响应的方式也有所不同。但是一般中断处理过程可分为 4 个阶段，即中断请求、中断响应、中断处理和中断返回。

AVR 单片机的中断向量区决定了不同的中断优先级，向量所在的地址越低，优先级越高：RESET 具有最高的优先级，INT0 次之，而 SPM_RDY 中断的优先级最低。当两个或多个中断源同时向 CPU 申请中断时，MCU 先响应优先级高的中断，低优先级的中断一般将保持中断标志位的状态（外部低电平中断除外），待优先级最高的中断服务程序执行完成返回后，再顺序响应优先级较低的中断。

MCU 响应一个中断后，在进入中断服务前已由硬件将全局中断允许位 I 自动清“0”。因此，此时即使有更高优先级的中断请求发生，MCU 也不会响应，要等执行到 RETI 指令，从本次中断返回，并执行了一条指令后，才能继续响应中断。因此，在默认情况下，AVR 的中断不能嵌套。如果需要中断嵌套，则需要在当前中断程序中加一条打开全局中断的语句，以人为地打开其他中断。

3．中断控制

ATmega16 有 3 个外部中断，即 INT0、INT1 和 INT2。INT0 和 INT1 中断的触发方式有下降沿触发、上升沿触发、任意电平变化触发、低电平触发，而 INT2 的触发方式仅为上升沿触发、下降沿触发，具体由外部中断控制寄存器的设置来确定。

在 ATmega16 中，与外部中断有关的寄存器主要有 SREG（状态寄存器）、MCUCR（MCU 控制寄存器）、MCUCSR（MCU 控制和状态寄存器）、GICR（通用中断控制寄存器）、GIFR（通用中断标志寄存器）。

【状态寄存器 SREG】状态寄存器 SREG 是一个 8 位标志寄存器，用于存放执行算术指令后的结果信息。它在 I/O 空间的地址为 0x3F（0x005F），它有 8 个标志位，但与中断有关的只有 I（bit 7）。其各位定义如下：

bit	7	6	5	4	3	2	1	0
	I	T	H	S	V	N	Z	C
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ I（bit7）：全局中断使能位。该标志位为 AVR 中断总控制开关，单独的中断使能由其他独立的寄存器控制。当 I 位置为“1”时，表示 CPU 可以响应中断请求；当 I 位清“0”时，不管单独中断标志位的状态是“0”还是“1”，都不会产生中断。当响应任意一个中断后，I 位由硬件清“0”，而执行 RETI 指令后，I 位将恢复置位为“1”以便重新使能中断。I 位也可以通过 SEI 和 CLI 指令进行软件置“1”和清“0”。

【MCU 控制寄存器 MCUCR】MCU 控制寄存器 MCUCR 包含中断触发控制位与通用 MCU 功能，在 I/O 空间的地址为 0x35（0x0055）。它有 8 个有效位，与中断有关的为低 4 位（bit3 ~ bit0），其高 4 位用于 ATmega16 单片机睡眠模式设定的使能等。其各位定义如下：

bit	7	6	5	4	3	2	1	0
	SM2	SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ ISC11、ISC10（bit3、bit2）：外部中断 1（INT1）中断触发方式选择位。

☺ ISC01、ISC00（bit1、bit0）：外部中断 0（INT0）中断触发方式选择位。

如果 SREG 的 I 位的相应中断屏蔽位置“1”时，可根据表 6-3 进行 INT1 和 INT0 的中断触发方式的选择。

表 6-3　INT1/INT0 的中断触发方式的选择

ISCx1	ISCx0	说　明
0	0	INTx（INT1 或 INT0）为低电平时产生中断请求
0	1	INTx（INT1 或 INT0）引脚上任意的逻辑电平变化都将引发中断
1	0	INTx（INT1 或 INT0）的下降沿时产生异步中断请求
1	1	INTx（INT1 或 INT0）的上升沿时产生异步中断请求

在检测边沿触发（上升沿触发或下降沿触发）前，MCU 会首先采样 INT0 引脚上的电平。如果选择了边沿触发方式或电平变化触发方式，那么持续时间大于一个时钟周期的脉冲将触发中断，过短的脉冲则不能保证触发中断。

🕮 注意：

如果使用外部中断低电平方式触发中断，应注意以下两点。

☺ 引脚（PD2、PD3）上的低电平必须一直保持到当前一条指令完成后才能触发中断。

☺ 低电平中断并不置位中断标志位，即外部低电平中断的触发不是由中断标志位引起的，而是由外部引脚上电平取反后直接触发中断的。因此，在使用低电平触发方式时，中断请求将一直保持到引脚的低电平消失为止。

【MCU 控制和状态寄存器 MCUCSR】MCU 控制和状态寄存器 MCUCSR 在 I/O 空间的地址为 0x34（0x0054）。它有 7 个有效位，与中断有关的为 ISC2 位（bit6），其他位用于单片机的复位控制。其各位定义如下：

bit	7	6	5	4	3	2	1	0
	JTD	ISC2	—	JTRF	WDRF	BORF	EXTRF	PORF
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ ISC2（bit6）：外部中断 2（INT2）的中断触发方式选择位。当 SREG 的 I 位的相应中断屏蔽位置“1”时，若 ISC2=0，则 INT2 的下降沿产生一个异步中断请求；若 ISC2=1，则 INT2 的上升沿产生一个异步中断请求。

INT2 的边沿触发方式（上升沿触发或下降沿触发）是异步的，只要 INT2 引脚（PB2）产生宽度超过 50ns 的脉冲就会引发中断。若选择了低电平中断，低电平必须保持到当前指令完成后才会产生中断。而且只要将引脚拉低，就会引发中断请求。改变 ISC2 时，有可能发生中断。因此建议首先在寄存器 GICR 里清除相应的中断使能位 INT2，然后再改变

ISC2。最后，不要忘记在重新使能中断前通过对 GIFR 寄存器的相应中断标志位 INTF2 写“1”使其清零。

【通用中断控制寄存器 GICR】通过设置通用中断控制寄存器 GICR 的相关位，可决定 INT0、INT1 和 INT2 中断是否使能。GICR 在 I/O 空间的地址为 0x3B（0x005B）。它有 5 个有效位，bit7～bit5 与中断有关。其各位定义如下：

bit	7	6	5	4	3	2	1	0
	INT1	INT0	INT2	—	—	—	IVSEL	IVCE
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ INT1（bit7）：外部中断 1（INT1）的中断允许控制位。当 INT1=1，且状态寄存器 SREG 的 I 位置“1”时，允许通过外部引脚 PD3 产生 INT1 中断。

☺ INT0（bit6）：外部中断 0（INT0）的中断允许控制位。当 INT0=1，且状态寄存器 SREG 的 I 位置“1”时，允许通过外部引脚 PD2 产生 INT0 中断。

☺ INT2（bit5）：外部中断 2（INT2）的中断允许控制位。当 INT2=1，且状态寄存器 SREG 的 I 位置“1”时，允许通过外部引脚 PB2 产生 INT2 中断。

【通用中断标志寄存器 GIFR】通用中断标志寄存器 GIFR 在 I/O 空间的地址为 0x3A（0x005A）。它有 3 个有效位，这 3 个有效位均与中断有关。其各位定义如下：

bit	7	6	5	4	3	2	1	0
	INTF1	INTF0	INTF2	—	—	—	—	—
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ INTF1（bit7）：外部中断 1（INT1）的中断标志位。

☺ INTF0（bit6）：外部中断 0（INT0）的中断标志位。

☺ INTF2（bit5）：外部中断 2（INT2）的中断标志位。

当 INT0、INT1 和 INT2 引脚上有效事件满足中断触发条件时，INTF0、INTF1 和 INTF2 位就会置“1”。如果此时 SREG 寄存器中的 I 位为“1”，且 GICR 寄存器中相应的中断使能位 INTx 为“1”，则 MCU 将响应中断请求，跳至相应的中断向量处开始执行中断服务程序，同时硬件自动将 INTFx 标志位清“0”。

> 📖 注意：
>
> 用户可以使用指令将 INTFx 清除，清除的方法是写逻辑“1”到 INTFx，将标志位清零。如果 INT1 和 INT0 设置为低电平触发方式，则标志位 INTF1 和 INTF0 始终为“0”，但这并不代表不产生中断请求，而只是因为低电平触发方式是不带中断标志类型的中断触发。在低电平触发方式时，中断请求将一直保持到引脚上的低电平消失为止。

4．CVAVR 程序中中断的使用

在 CVAVR 中，中断服务程序必须定义成一个特殊的函数，此函数称为中断服务函数。中断服务函数的定义格式如下：

```
interrupt [中断向量号] void  函数名 (void)
{
```

```
    //函数体;
}
```

其中，关键字 interrupt 声明了该函数为中断服务函数，以区别于一般软件调用的函数。[中断向量号]表明该函数属于哪一个中断服务函数（可参考表 6-1）。由于中断函数是 MCU 响应中断时通过硬件自动调用的，所以中断函数的返回值和参数均为 void（不能返回函数值，也不带参数）。函数名可由用户自行定义，但要符合命名规则。

按照上面的格式，定时器/计数器 1 溢出时（TIMER1 OVF，中断向量号为 9）的中断服务函数可以定义如下：

```
interrupt [9] void    timer0_overflow (void)
{
    //函数体;
}
```

中断向量号也可以用头文件“mega16.h”中定义的宏来代替。例如，外部中断 INT1（3 号中断）的中断服务函数可以定义如下：

```
interrupt [EXT_INT1] void    int1_isr (void)
{
    //函数体;
}
```

其中，EXT_INT1 是在头文件“mega16.h”中定义的宏，等同于数字 2。为了便于中断服务程序的编写和阅读，CVAVR 对 ATmega16 中断源的宏定义见表 6-4。对于其他型号 AVR 单片机的中断源，CVAVR 也定义了类似的宏，详细定义请参考各个型号 AVR 单片机的头文件（包含在 CVAVR 的 inc 子目录下）。

表 6-4　CVAVR 对 ATmega16 中断源的宏定义

中　断　源	宏　定　义	中　断　源	宏　定　义
INT0	#define EXT_INT0 2	USART RXC	#define USART_RXC 12
INT1	#define EXT_INT1 3	USART UDRE	#define USART_DRE 13
TIMER2 COMP	#define TIM2_COMP 4	USART TXC	#define USART_TXC 14
TIMER2 OVF	#define TIM2_OVF 5	ADC	#define ADC_INT 15
TIMER1 CAPT	#define TIM1_CAPT 6	EE_RDY	#define EE_RDY 16
TIMER1 COMPA	#define TIM1_COMPA 7	ANA_COMP	#define ANA_COMP 17
TIMER1 COMPB	#define TIM1_COMPB 8	TWI	#define TWI 18
TIMER1 OVF	#define TIM1_OVF 9	INT2	#define EXT_INT2 19
TIMER0 OVF	#define TIM0_OVF 10	TIMER0 COMP	#define TIM0_COMP 20
SPI STC	#define SPI_STC 11	SPM_RDY	#define SPM_READY 21

编译器在中断调用时会自动保存所有使用的寄存器，并在中断返回时恢复它们的值。中断结束的地方是一条 RETI 指令。中断没有返回值，也不带参数。使用中断前，必须设置相关的控制寄存器中的位以允许中断。

默认情况下，CVAVR 在生成中断服务函数时，会自动将 R0, R1, R15, R22, R27, R30, R31, SREG 及用户中断服务程序中所要使用的所有通用寄存器保护起来。如果用户想关闭或

打开 CVAVR 的中断现场保护与恢复功能时，可使用#pragrma savereg 指令来实现。例如：

```
#pragram savereg-      //关闭寄存器自动保存和恢复（即关闭自动现场保护与恢复）功能
interrupt [EXT_INT1]   void int1_isr (void)
{
     /*只保存使用的寄存器 R23、R24 和 SREG*/
    #asm
      push   r23
      push   r24
      in r23,SREG
    #endasm
    /*编写 C 语言程序代码*/
    /*……*/
    /*恢复 SREG、R24、R23*/
     #asm
      pop   r23
      out   SREG,r23
      pop   r24
      pop   r23
    #endasm
}
  #pragma savereg+   //打开寄存器自动保存和恢复（即重新开放自动生成现场保护与恢复）功能
```

在 CVAVR 中使用 C 语言编写系统程序时，可按以下程序框架进行：

```
#include "mega16.h"
//***********************第 1 部分:宏定义及函数的编写*******************************
//相关宏定义
//编写一般函数，如 delay 延时函数、display 显示函数
interrupt [EXT_INT1 void   int0_isr (void)    //中断服务函数（在此以 INT1 为例）
{
     ┆                                        //CPU 响应 INT1 时完成的任务
}
    //***********************第 2 部分:主程序的编写*********************************
   void main (void)
     {
          ┆
         //INT1 的初始化
           MCUCR=0x08;                       //INT1 为下降沿时产生中断请求
           GICR=0x80;                        //允许 INT1 产生中断
           #asm("sei")                       //使能全局中断
          While(1)
            {
              ┆                              //正常程序完成的任务
             }
     }
```

程序框架中的#asm("sei")表示使能全局中断，与 SREG|=0x80（即将 I 位置“1”）等价。用#asm("cli")可以禁止全局中断。

> 注意:
> 使用中断函数时要注意下述问题。

☺ 在设计中断时，要注意哪些功能应该放在中断服务程序中，哪些功能应放在主程序中。一般来说，中断服务程序应该做最少量的工作，这样做有很多好处。首先，系统对中断的反应面更宽了，有些系统如果丢失中断或中断响应太慢将产生十分严重的后果，这时有充足的时间等待中断是十分重要的。其次，它可使中断服务程序的结构简单，不容易出错。中断程序中放入的东西越多，它们之间越容易发生冲突。简化中断服务程序意味着软件中将有更多的代码段，但可把这些都放入主程序中。中断服务程序的设计对系统的成败有至关重要的作用，要仔细考虑各中断之间的关系和每个中断执行的时间，特别要注意那些对同一个数据进行操作的中断服务程序（Interrupt Service Routine，ISR）。

☺ 中断函数不能传递参数，没有返回值。

☺ 中断函数调用其他函数时，要保证使用相同的寄存器组，否则将出错。

☺ 中断函数使用浮点运算时，要保证浮点寄存器的状态。

任务 2　带中断控制的 LED 花样灯

PC 端口作为输出端口，外接 8 只 LED D0～D7，PD2 外接按键 K。编写程序，当按键 K 未被按下时，8 只 LED 进行默认显示，其显示规律为左移流水→右移流水→隔灯闪烁 3 次花样显示；如果按下按键 K 后，8 只 LED 暂停默认显示，而进行闪烁显示，当显示达到 5 次后，继续按默认规律显示。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 6-5 所列的元器件，并完成图 6-1 所示的硬件电路图设计。

表 6-5　带中断控制的 LED 花样灯所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	电解电容 CAP-ELEC	发光二极管 LED-BIBY
电阻 RES	晶振 CRYSTAL 8MHz	发光二极管 LED-BIRG	发光二极管 LED-YELLOW
按钮 BUTTON	发光二极管 LED-BIGY		

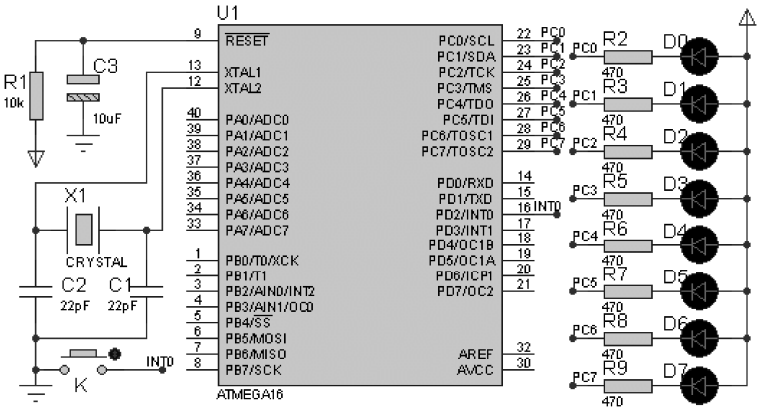


图 6-1　带中断控制的 LED 花样灯电路图

软件设计

一般来说，按键处理方式有两种：第 1 种方法是不断地查询按键，有按键被按下时进行消抖处理，再判断是否真的有按键被按下，这种方法在按键查询期间不能做任何其他操作，如在项目五中任务 6 的按键处理就采用查询方式；第 2 种方法就是采用外部中断方式，当按键被按下时就执行按键被按下的任务，否则执行其他任务。

本任务采用中断方式进行，当没有按键被按下时，LED 就进行默认的花样显示，按下按键时就执行闪烁显示。默认花样显示，在此可以通过建立一个 discode []数组来实现。要完成中断操作，在主程序中应该先进行中断初始化设置，包括中断方式设置、中断启动等。本设计采用 INT0（外部中断 0），根据外部中断相应的寄存器可知，使用 INT0 时，MCUCR=0b00000010=0x02，即 INT0 采用下降沿触发方式；GICR=0b01000000=0x60，即允许 INT0 产生中断请求。要将中断总允许位 I 置为“1”时，可使用#asm("sei")指令，也可采用 SERG=0x80 来实现（0x80=0b10000000）。

中断函数中为保证按键真的被按下也可以采用软件消抖方式来完成，如果真的被按下，则进行闪烁显示控制，对于保护现场、恢复现场和中断返回，在中断服务函数可以不需写出相应的指令。带中断控制的 LED 花样灯程序流程图如图 6-2 所示。

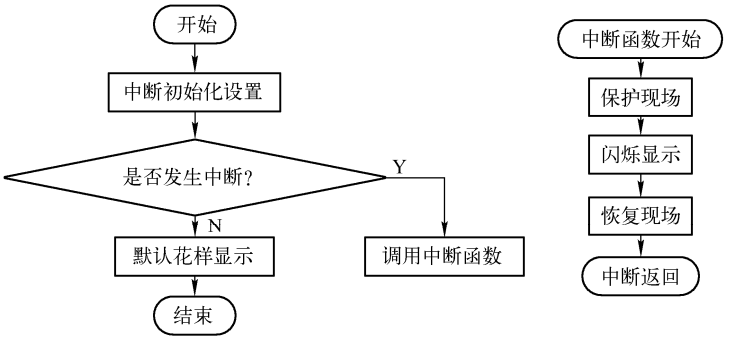


图 6-2　带中断控制的 LED 花样灯程序流程图

源程序

```
/*****************************************************
  文件名:          带中断控制的 LED 花样灯.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define   button          PIND.2
#define   LED               PORTC
flash uchar discode1[]={
            0xFE,0xFD,0xFB,0xF7,0xEF,0xDF,0xBF,0x77,          //左移流水灯
            0xBF,0xDF,0xEF,0xF7,0xFB,0xFD,0xFE,0xFF,          //右移流水灯
            0xAA,0x55,0xAA,0x55,0xAA,0x55,0xFF };             //隔灯闪烁
```

```
flash uchar discode2[]={0xFF,0x00,0xFF,0x00,0xFF,
                0x00,0xFF,0x00,0xFF,0x00};                    //闪烁 5 次
void delay(uint k)
{
   uint m,n;
   for(m=0;m<k;m++)
      {
         for(n=0;n<1140;n++)
           {;}
      }
}
interrupt[EXT_INT0] void int0_isr(void)                       //INT0 中断函数
{
   uchar i;
   delay(10);
   if(button==0)                                              //再次确认按键，软件防抖
      {
         for(i=0;i<10;i++)
            {
         LED=discode2[i];
         delay(500);
             }
      }
}
void port_init(void)                                          //端口初始化
{
      DDRC=0xFF;
     PORTC=0xFF;
      DDRD=0xFB;
     PORTD=0xFF;
}
void INT0_init(void)                                          //中断初始化设置
{
     MCUCR=0x02;                                              //INT0 为下降沿时产生中断请求
      GICR=0x60;                                              //允许 INT0 产生中断
      #asm("sei")                                             //使能全局中断
}
void main(void)
{
   port_init();
   INT0_init();
   while(1)
      {
          uchar i;
          for(i=0;i<23;i++)
           {
              LED=discode1[i];
               delay(500);
```

```
        }
    }
}
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“带中断控制的 LED 花样灯.COFF”文件，然后在 Proteus ISIS 中打开已创建的带中断控制的 LED 花样灯电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在没有按下按键时，8 只 LED 进行默认显示，按下按键后，8 只 LED 暂停默认显示，转而执行闪烁显示，当闪烁显示达设定次数后，继续默认显示。带中断控制的 LED 花样灯的运行仿真效果如图 6-3 所示。

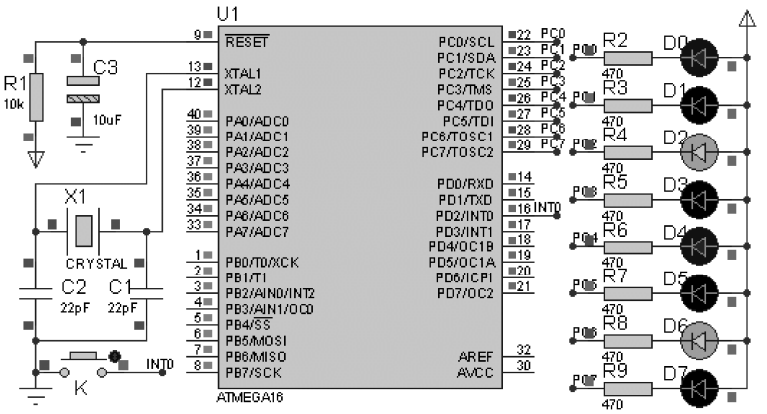


图 6-3　带中断控制的 LED 花样灯运行仿真效果图

任务 3　采用外中断控制的条形 LED 彩灯设计

PC 端口作为输出端口，外接条形 LED，PD3 外接按键 K。编写程序，当按键 K 未被按下时，条形 LED 进行循环左移显示。如果第 1 次按下按键 K 后，条形 LED 进行循环右移显示。如果第 2 次按下按键 K 后，条形 LED 进行拉幕式与闭幕式花样显示。如果第 3 次按下按键 K 后，又恢复循环左移显示。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 6-6 所列的元器件，并完成图 6-4 所示的硬件电路图设计。

表 6-6　采用外中断控制的条形 LED 彩灯所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	电解电容 CAP-ELEC	按钮 BUTTON
电阻 RES	晶振 CRYSTAL 8MHz	条形发光二极管 LED-BARGRAPH-GRN	

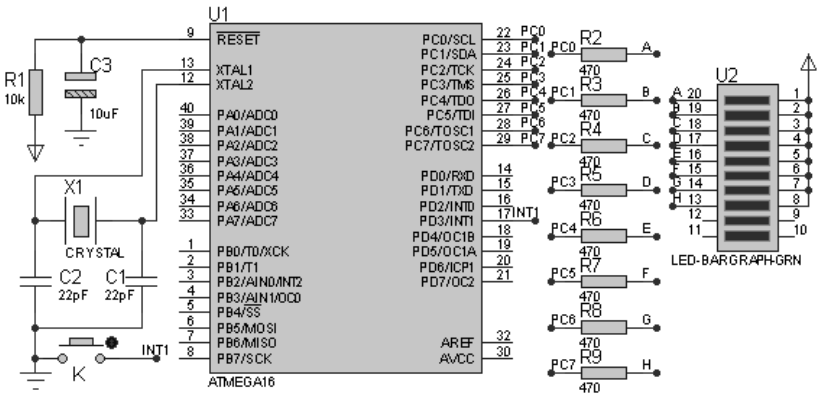


图 6-4　采用外中断控制的条形 LED 彩灯电路图

软件设计

本设计只用一个按键就能完成多个任务的选择控制，对于一键多任务的选择控制，使用中断方式进行可能是较好的方式。

在主程序中需进行 INT1 的初始化。根据外部中断相应的寄存器可知，使用 INT1 时，MCUCR=0b00001000=0x08，即 INT1 采用下降沿触发方式；GICR=0b10000000=0x80，即允许 INT1 产生中断请求。要将中断总允许位 I 置为“1”时，可使用#asm("sei")指令，也可采用 SERG=0x80 来实现（0x80=0b10000000）。

在中断函数中使用变量 a 来实现一键多任务的选择控制，在未按下按键 K 时（即没有中断请求），a 为默认初值，每次按下按键 K 时，a 加 1，当 a 加到一定时，a 又回到初值，然后再次按下按键 K 时，a 又加 1。在主程序中，根据 a 的值而执行相应的功能。采用外中断控制的条形 LED 彩灯的程序流程图如图 6-5 所示。

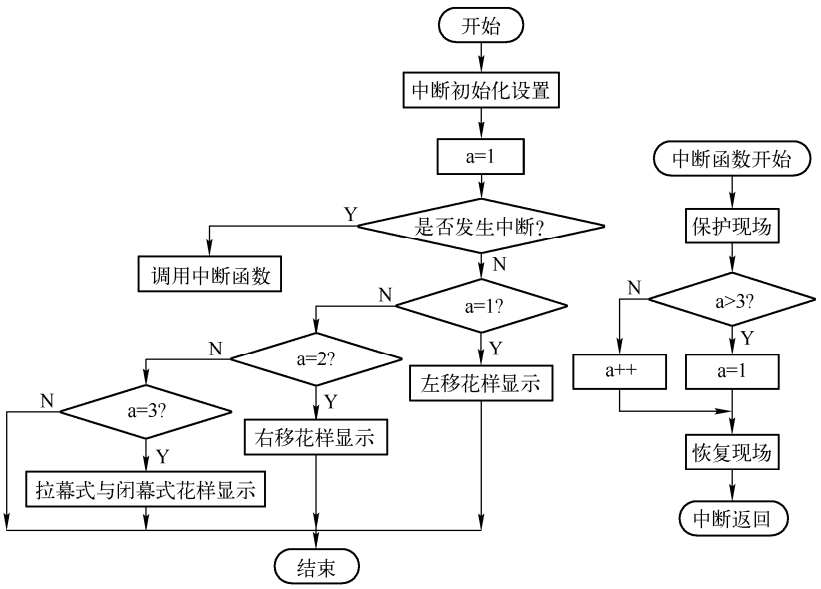


图 6-5　采用外中断控制的条形 LED 彩灯程序流程图

PROGRAM 源程序

```
/****************************************************
 文件名:          采用外中断控制的条形 LED 彩灯.c
 单片机型号:      ATmega16
 晶振频率:        8.0MHz
****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define   button          PIND.3
#define   LED               PORTC
flash uchar discode1[]={0xFE,0xFD,0xFB,0xF7,0xEF,0xDF,0xBF,0x7F};          //左移流水
flash uchar discode2[]={0xBF,0xDF,0xEF,0xF7,0xFB,0xFD,0xFE,0xFF};          //右移流水
flash uchar discode3[]={0x18,0x24,0x42,0x81,0x7E,0x3C,0x18,0x00};          //拉幕式与闭幕式
uchar a;
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void display1(void)
{
  uchar i;
  for(i=0;i<8;i++)                          //D0～D7 移位点亮（左移）
      {
          LED=discode1[i];
          delay(500);
      }
}
void display2(void)
{
  uchar i;
  for(i=0;i<8;i++)                          //D7～D0 移位点亮（右移）
      {
          LED= discode2[i];
          delay(500);
      }
}
void display3(void)
{
  uchar i,j;
  for(i=0;i<8;i++)                          //拉幕式与闭幕式
      {
```

```
            j=discode3[i];
            LED=~j;
            delay(500);
        }
}
interrupt[EXT_INT1] void int1_isr(void)                    //INT1 中断函数
{
   delay(10);
   if(button==0)                                           //再次确认按键，软件防抖
      {
          a++;
          if(a>3)
             {
                a=1;
             }
      }
}
void port_init(void)                                       //端口初始化
{
     DDRC=0xFF;
    PORTC=0xFF;
     DDRD=0xF7;
    PORTD=0xFF;
}
void INT1_init(void)                                       //中断初始化设置
{
    MCUCR=0x08;                                            //INT1 为下降沿时产生中断请求
     GICR=0x80;                                            //允许 INT1 产生中断
    SREG|=0x80;                                            //使能全局中断   #asm("sei")
}
void main(void)
{
   port_init();
   INT1_init();
   a=1;
   while(1)
      {
          switch(a)
                {
                   case 1:display1();break;
                  case 2:display2();break;
                  case 3:display3();break;
                }
        }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“采用外中断控制的条形 LED 彩

灯.COFF”文件，然后在 Proteus ISIS 中打开已创建的采用外中断控制的条形 LED 彩灯电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在没有按下按键时，条形 LED 进行循环左移显示。如果第 1 次按下按键 K 后，条形 LED 进行循环右移显示。如果第 2 次按下按键 K 后，条形 LED 进行拉幕式与闭幕式花样显示。如果第 3 次按下按键 K 后，又恢复循环左移显示。采用外中断控制的条形 LED 彩灯的运行仿真效果如图 6-6 所示。

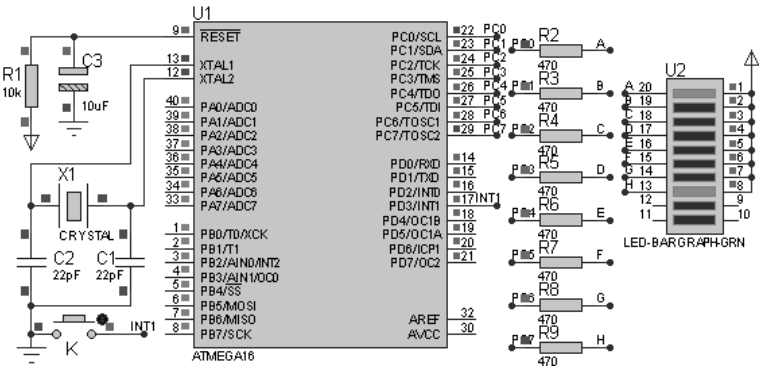


图 6-6　采用外中断控制的条形 LED 彩灯仿真效果图

任务 4　采用两个外中断的条形 LED 彩灯控制

PC 端口作为输出端口，外接条形 LED，PD2 外接按键 K1，PD3 外接按键 K2。编写程序，没有按下任何按键时，条形 LED 进行隔灯闪烁；当按下按键 K1 为奇数次时，条形 LED 进行左移流水→右移流水显示；当按下按键 K1 为奇数次时，条形 LED 进行拉幕式显示；当按下按键 K2 时，条形 LED 暂停显示，蜂鸣器发出“叮咚”的开门声音，然后条形 LED 继续花样显示。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 6-7 所列的元器件，并完成图 6-7 所示的硬件电路图设计。

表 6-7　采用两个外中断的 LED 条形彩灯控制所用元器件

单片机 ATmega16	电解电容 CAP-ELEC	按钮 BUTTON	电阻 RES	蜂鸣器 SOUNDER
瓷片电容 CAP 22pF	晶振 CRYSTAL 8MHz	条形发光二极管 LED-BARGRAPH-GRN		

软件设计

本任务使用了两个外部中断，按键 K1 为外部中断 0 的中断源；按键 K2 为外部中断 1 的中断源。在编写程序时，需要编写两个中断函数。蜂鸣器发出“叮咚”的开门声音，其原理是控制 PD0 输出一段占空比不同的脉冲。采用两个外中断的条形 LED 彩灯控制程序流程

图如图 6-10 所示。

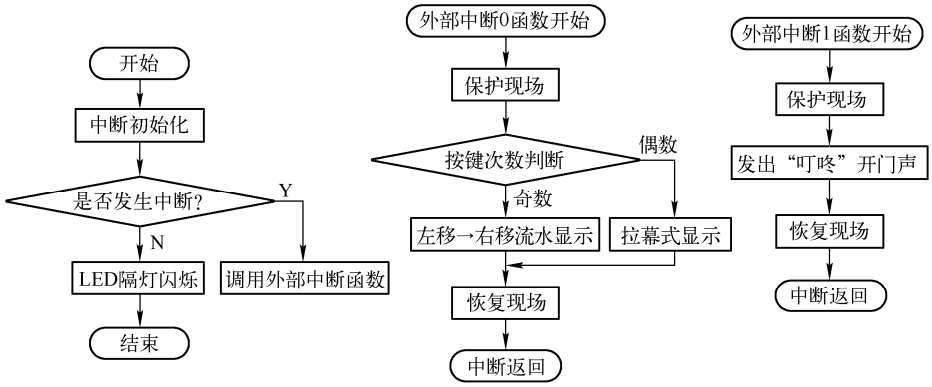


图 6-7　采用两个外中断的条形 LED 彩灯控制程序流程图

源程序

```
/****************************************************
  文件名:          采用两个外中断的条形 LED 彩灯控制.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define   MIC              PIND.0
#define   button1          PIND.2
#define   button2          PIND.3
#define   LED              PORTC
flash uchar discode1[]={0xFE,0xFD,0xFB,0xF7,0xEF,0xDF,0xBF,0x7F,          //左移流水
                        0xBF,0xDF,0xEF,0xF7,0xFB,0xFD,0xFE,0xFF};         //右移流水
flash uchar discode2[]={0x18,0x24,0x42,0x81};                             //拉幕式
flash uchar discode3[]={0xAA,0x55,0xAA,0x55,0xAA,0x55,0xFF};              //隔灯闪烁
uchar a;
bit    sp;
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
interrupt[EXT_INT0] void int0_isr(void)                                   //INT0 中断函数
{
  delay(10);
```

```
    if(button1==0)                                    //再次确认按键，软件防抖
      {
        a++;
        if (a==1)
          {
            uchar i;
            for(i=0;i<16;i++)
              {
                  LED=discode1[i];
                 delay(500);
              }
          }
        if (a==2)
          {
            uchar i;
            for(i=0;i<4;i++)
              {
                  LED=discode2[i];
                  delay(500);
              }
            a=0;
          }
    }
}
interrupt[EXT_INT1] void int1_isr(void)               //INT1 中断函数
{
   delay(10);
   if(button2==0)                                     //再次确认按键，软件防抖
      {
        uchar i;
        for(i=0;i<200;i++)
          {
            sp=~sp;
            MIC=sp;
            delay(1);
          }
        for(i=0;i<200;i++)
          {
            sp=~sp;
            MIC=sp;
            delay(2);
          }
      }
}
void port_init(void)                                  //端口初始化
{
     DDRC=0xFF;
    PORTC=0xFF;
```

```
    DDRD=0xF3;
   PORTD=0xFF;
}
void INT_init(void)                          //中断初始化设置
{
   MCUCR=0x0A;                               //定义 INT0 和 INT1 为下降沿时产生中断
    GICR=0xC0;                               //允许 INT0 和 INT1 产生中断
    #asm("sei")                              //开启总中断
}
void main(void)
{
  uchar i;
  port_init();
  INT_init();
  a=0;
  while(1)
    {
      for(i=0;i<7;i++)
        {
          LED=discode3[i];
          delay(500);
        }
    }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“采用两个外中断的条形 LED 彩灯控制.COFF”文件，然后在 Proteus ISIS 中打开已创建的采用两个外中断的条形 LED 彩灯控制电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在按键 K1 和 K2 都没有被按下时，条形 LED 进行隔灯闪烁。如果按下按键 K1 为奇数次时，条形 LED 进行左移流水→右移流水显示，然后再回到隔灯闪烁状态。如果按下按键 K1 为偶数次时，条形 LED 进行拉幕式显示，然后再回到隔灯闪烁状态。如果按下按键 K2 时，条形 LED 暂停显示，蜂鸣器发出“叮咚”的开门声音，然后条形 LED 继续花样显示。采用两个外中断的条形 LED 彩灯控制的运行仿真效果如图 6-8 所示。

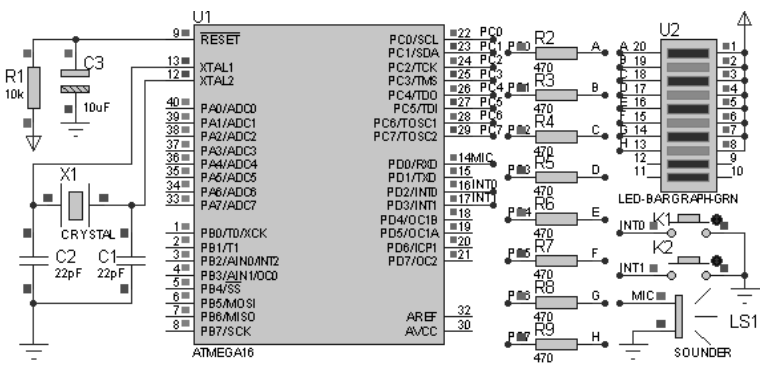


图 6-8　采用两个外中断的条形 LED 彩灯控制仿真效果图

任务 5　采用两个外中断实现 LED 键控移位

PC 端口作为输出端口，外接 8 只 LED，PD2 外接按键 K1，PD3 外接按键 K2。编写程序，要求实现 LED 键控移位。即按下按键 K1 时，8 只 LED 左移 1 位；按下按键 K2 时，8 只 LED 左移 1 位。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 6-8 所列的元器件，并完成图 6-9 所示的硬件电路图设计。

表 6-8　采用两个外中断实现 LED 键控移位所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	电解电容 CAP-ELEC	发光二极管 LED-BIBY
电阻 RES	晶振 CRYSTAL 8MHz	发光二极管 LED-BIRG	发光二极管 LED-YELLOW
按钮 BUTTON	发光二极管 LED-BIGY		

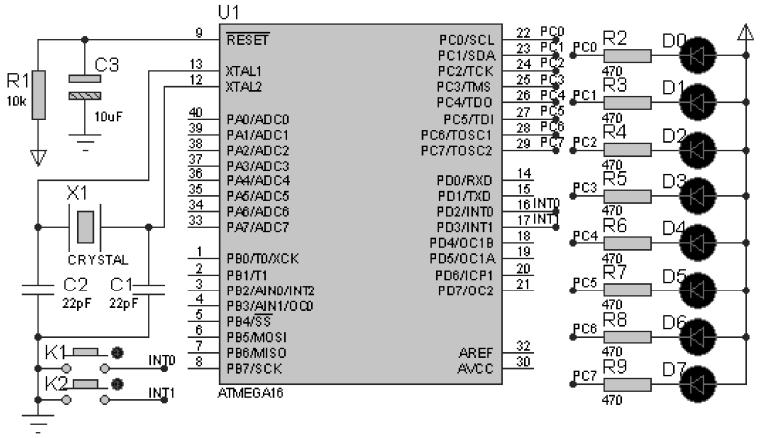


图 6-9　采用两个外中断实现 LED 键控移位电路图

软件设计

所谓键控移位，就是按下 1 次按键时，LED 向左或向右移动 1 位，没有按下时状态保持。本任务也使用了两个外部中断，按键 K1 为外部中断 1 的中断源；按键 K2 为外部中断 2 的中断源。按键 K1 实现键控左移，按键 K2 实现键控右移时，可以定义两个变量 a、b。由于图 6-9 中各 LED 是低电平点亮，因此，变量 a 实现左移或右移后，将数据送给变量 b，然后 b 中的内容取反送给 PC 端口即可实现 LED 的状态显示。采用两个外中断实现 LED 键控移位的程序流程图如图 6-10 所示。

在主程序中需进行 INT0 和 INT1 的初始化。根据外部中断相应的寄存器可知，使用 INT0 和 INT1 时，MCUCR=0b00001010=0x0A，即 INT0、INT1 采用下降沿触发方式；GICR=0b11000000=0xC0，即允许 INT0 和 INT1 产生中断请求。要将中断总允许位 I 置为“1”时，可使用#asm("sei")指令，也可采用 SERG=0x80 来实现（0x80=0b10000000）。

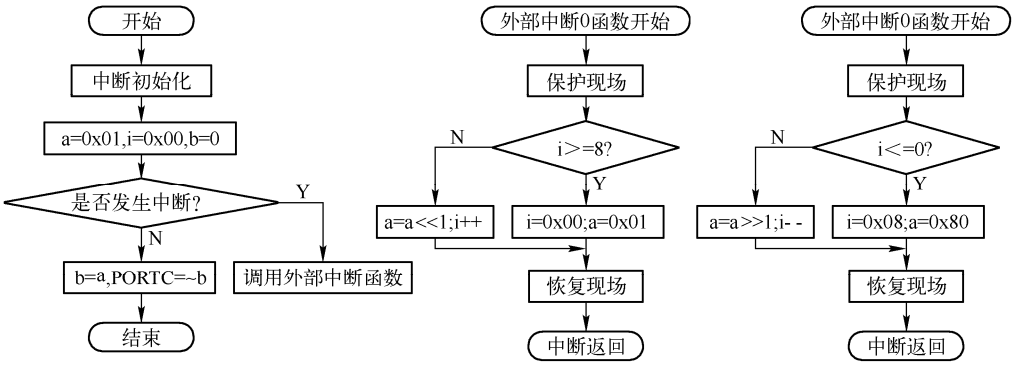


图 6-10　采用两个外中断实现 LED 键控移位程序流程图

源程序

```
/*****************************************************
  文件名:          采用两个外中断实现 LED 键控移位.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#define   uchar unsigned char
#define   uint unsigned int
#define   button1          PIND.2
#define   button2          PIND.3
#define   LED              PORTC
uchar a,i;
void delay(uint k)
{
   uint m,n;
   for(m=0;m<k;m++)
      {
         for(n=0;n<1140;n++)
           {;}
      }
}
interrupt[EXT_INT0] void int0_isr(void)                    //INT0 中断函数
{
   delay(10);
   if(button1==0)                                          //再次确认按键，软件防抖
      {
         if(i>=8)
            {
               i=0x00;
               a=0x01;
            }
         else
            {
```

```
            i++;
            a=a<<1;
          }
      }
}
interrupt[EXT_INT1] void int1_isr(void)          //INT1 中断函数
{
  delay(10);
  if(button2==0)                                 //再次确认按键，软件防抖
    {
      if(i<=0)
        {
          i=0x08;
          a=0x80;
        }
      else
        {
          i--;
          a=a>>1;
        }
    }
}
void port_init(void)                             //端口初始化
{
   DDRC=0xFF;
  PORTC=0xFF;
   DDRD=0xF3;
  PORTD=0xFF;
}
void INT_init(void)                              //中断初始化设置
{
  MCUCR=0x0A;                                    //定义 INT0 和 INT1 为下降沿时产生中断
   GICR=0xC0;                                    //允许 INT0 和 INT1 产生中断
   #asm("sei")                                   //开启总中断
}
void main(void)
{
  uchar b;
  port_init();
  INT_init();
  i=0x00;
  a=0x01;
  b=0x00;
  while(1)
    {
      b=a;
```

```
            LED=~b;
        }
    }
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“采用两个外中断实现 LED 键控移位.COFF”文件，然后在 Proteus ISIS 中打开已创建的采用两个外中断实现 LED 键控移位电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在按键 K1、按键 K2 都没有被按下时，LED D0 亮。如果按下按键 K1 时，LED 左移 1 位显示；如果按下按键 K2 时，LED 右移 1 位显示。采用两个外中断实现 LED 键控移位的运行仿真效果如图 6-11 所示。

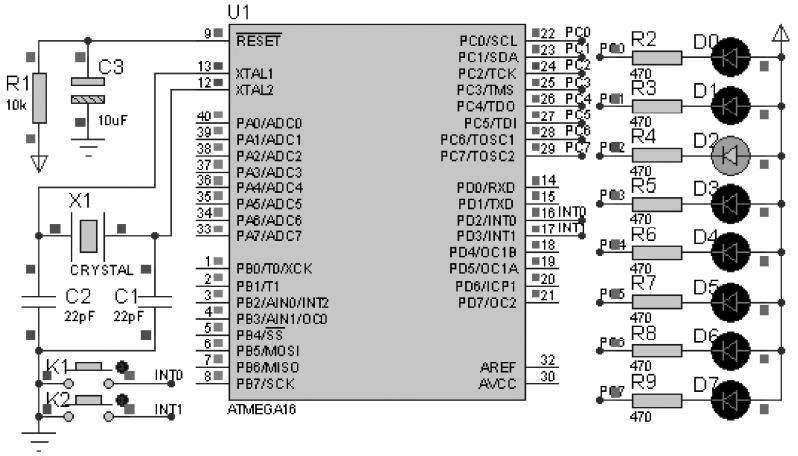


图 6-11　采用两个外中断实现 LED 键控移位仿真效果图

任务 6　采用外中断实现计数

使用单片机 PC、PB0、PB1 端口作为输出端口，外接一个 2 位 LED 数码管，PD2 外接按键 K1，PB2 外接按键 K2。编写程序，每次按键 K1 被按下时，LED 数码管进行加 1 显示；每次按下按键 K2 时，LED 数码管进行减 1 显示，计数范围为 0～99。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 6-9 所列的元器件，并完成图 6-12 所示的硬件电路图设计。

表 6-9　采用外中断实现计数所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	电解电容 CAP-ELEC	按钮 BUTTON
电阻 RES	晶振 CRYSTAL 8MHz	数码管 7SEG-MPX2-CA-BULE	三极管 NPN

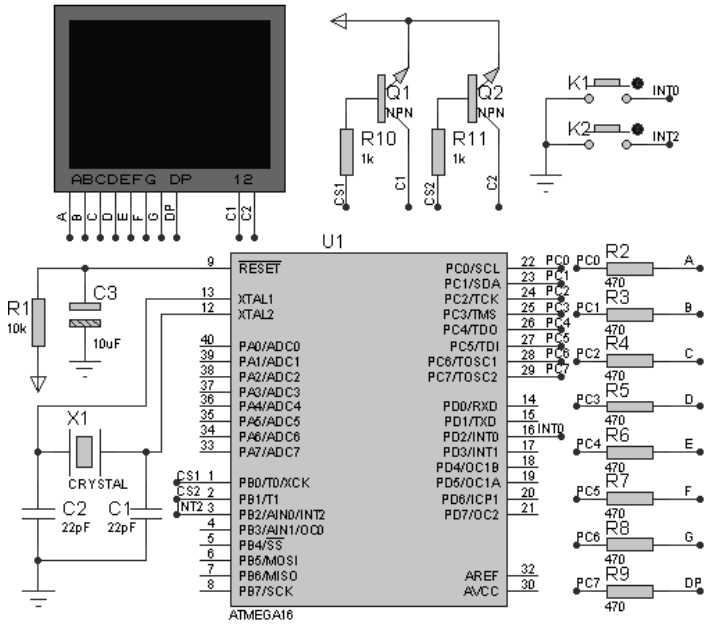


图 6-12　采用外中断实现计数电路图

软件设计

本设计也使用了两个外部中断，按键 K1 控制加 1 计数；按键 K2 控制减 1 计数。使用变量 a 作为加 1 或减 1，当按键 K1 被按下时，首先判断 a 是否为 99，如果是 99 则将 a 清零，否则 a 加 1；当按键 K2 被按下时，首先判断 a 是否为 0，如果是 0 则将 a 置为 99，否则减 1。

由于 LED 数码管采用动态扫描显示两位数据，因此对 a 需进行十位与个位的分离，将分离后的十位数字送给 data_H；分离后的个位数字送给 data_L。当 CS1 有效时，PORTC=tab[data_H]；CS2 有效时，PORTC=tab[data_L]。C1、C2 的有效切换较短，在 LED 数码管上即可动态显示 0～99。采用外中断实现计数的程序流程图如图 6-13 所示。

在主程序中，需进行 INT0 和 INT2 的初始化。根据外部中断相应的寄存器可知，使用 INT0 和 INT2 时，MCUCR=0b00000010=0x02，MCUCSR=0b00000000=0x00，即 INT0、INT2 采用下降沿触发方式；GICR=0b01100000=0x60，即允许 INT0 和 INT2 产生中断请求。要将中断总允许位 I 置为“1”时，可使用#asm("sei")指令，也可采用 SERG=0x80 来实现（0x80=0b10000000）。

源程序

```
/*****************************************************
  文件名:          采用外中断实现计数.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#define   uchar unsigned char
```

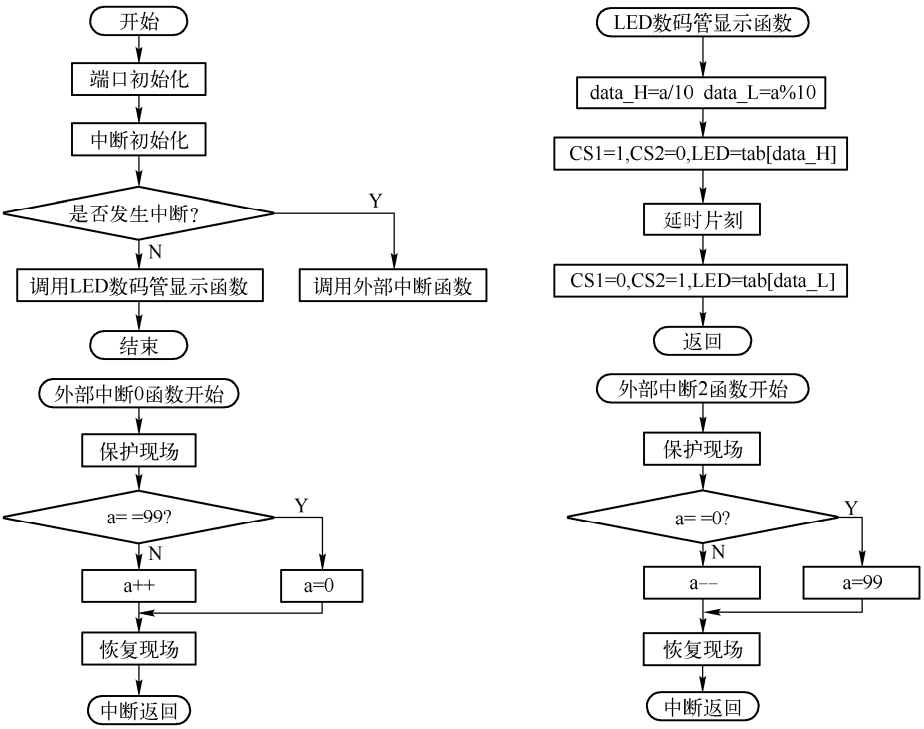


图 6-13　采用外中断实现计数程序流程图

```
#define   uint unsigned int
#define   button1          PIND.2
#define   button2          PINB.2
#define   CS1               PORTB.0                    //CS1 片选
#define   CS2               PORTB.1                    //CS2 片选
#define   LED               PORTC
flash uchar tab[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,  //共阳极 LED0～F 的段码
                   0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E,0xBF,0x10};
uchar a;                                              //计数值
uchar data_H,data_L;                                  //计数值高、低位
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
interrupt[EXT_INT0] void int0_isr(void)               //INT0 中断函数
{
  delay(10);
  if(button1==0)                                      //再次确认按键，软件防抖
    {
```

```
        if(a==99)
          {
            a=0;
          }
        else
          {
            a++;
          }
      }
}
interrupt[EXT_INT2] void int2_isr(void)                    //INT2 中断函数
{
  delay(10);
  if(button2==0)                                           //再次确认按键，软件防抖
    {
      if(a==0)
        {
          a=99;
        }
      else
        {
          a--;
        }
    }
}
void data_in(void)                                         //获取高、低位
{
  data_L=a%10;
  data_H=a/10;
}
void display(void)
{
  LED=tab[data_H];
  CS1=1;
  delay(5);
  CS1=0;
  LED=tab[data_L];
  CS2=1;
  delay(5);
  CS2=0;
}
void port_init(void)                                       //端口初始化
{
     DDRB=0xFB;                                            //PB 端口配置为输出
   PORTB=0xFF;                                             //PB 端口初始值设置
     DDRC=0xFF;                                            //PC 端口配置为输出
   PORTC=0xFF;                                             //PC 端口初始值设置
     DDRD=0xFB;                                            //PD 端口配置为输出
   PORTD=0xFF;                                             //PD 端口初始值设置
}
```

```
void INT_init(void)                          //中断初始化设置
{
    MCUCR=0x02;                              //INT0 为下降沿时产生中断请求
    MCUCSR=0x00;                             //INT2 为下降沿时产生中断请求
     GICR=0x60;                              //允许 INT0 和 INT2 产生中断
     #asm("sei")                             //开启总中断
}
void main(void)
{
  port_init();
  INT_init();
  a=0;
  while(1)
     {
       data_in();
       display();
     }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“采用外中断实现计数.COFF”文件，然后在 Proteus ISIS 中打开已创建的采用外中断实现计数电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在按键 K1、K2 都没有被按下的初始状态时，LED 数码管显示 0。如果按下按键 K1 时，LED 数码管进行加 1 计数，当计数值达到 99 时，再次按下该按键则显示为 0；如果按下按键 K2 时，LED 数码管进行减 1 计数，当计数值减到 0 时，再次按下该按键则显示为 99。采用外中断实现计数的运行仿真效果如图 6-14 所示。

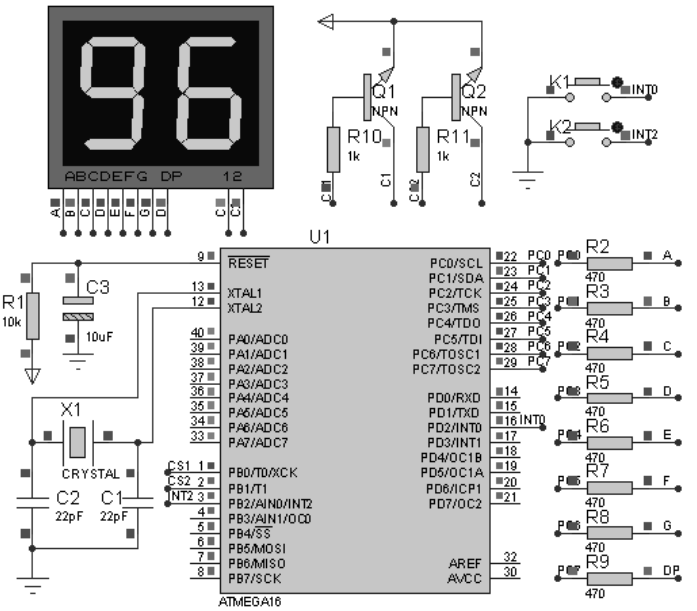


图 6-14　采用外中断实现计数仿真效果图

任务 7　采用外中断进行计数和清零控制

使用单片机 PC、PB0、PB1 端口作为输出端口，外接一个 2 位 LED 数码管，PD2 外接按键 K1，PD3 外接按键 K2，PB2 外接按键 K3。编写程序，按键 K3 作为“开始/暂停/清零”控制；按键 K1 作为加 1 计数控制，按键 K2 作为减 1 计数控制。当按键 K2 没有被按下时，按键 K1、K2 被按下后，LED 数码管总显示为 0；第 1 次按下按键 K2 时，每次按下按键 K1，LED 数码管进行加 1 显示，每次按下按键 K2，LED 数码管进行减 1 显示；第 2 次按下按键 K3 时，暂停 0.1s 加 1/减 1 计数，按下按键 K1 或 K2 后，LED 数码管显示暂停前的数值；第 3 次按下按键 K3 时，LED 数码管显示为 0；第 4 次按下按键 K3 时，每次按下按键 K1 或 K2，LED 数码管进行加 1/减 1 显示。

硬件设计

本设计所使用的元器件与表 6-9 相同，其硬件电路图如图 6-15 所示。

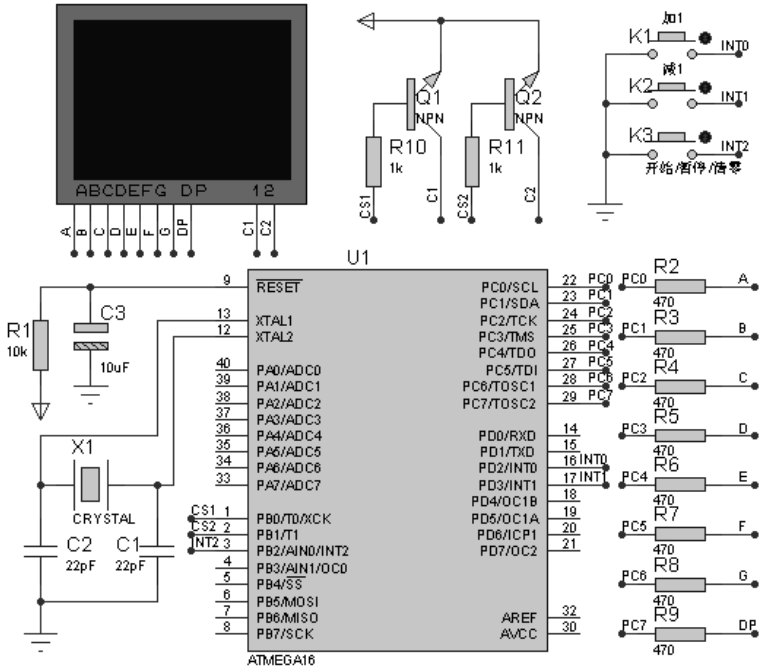


图 6-15　采用外中断进行计数和清零控制电路图

软件设计

本设计使用了 3 个外部中断，按键 K1 控制加 1 计数；按键 K2 控制减 1 计数；按键 K3 控制“加 1/暂停/清零”，所以外部中断 0、外部中断 1 由外部中断 2 控制开启与关闭。使用变量 a 和 b 作为加 1/减 1 控制，其中 a 作为 LED 显示的计数变量；b 作为按下按键 K2 次数统计的变量。

在外部中断函数 0 中，当按下按键 K1 时，首先判断 a 是否为 99，如果是 99 则将 a 清零，否则 a 加 1。在外部中断函数 1 中，当按下按键 K2 时，首先判断 a 是否为 0，如果是 0

则将 a 置为 99，否则 a 减 1。

在外部中断 2 函数中，当按下按键 K3 时，b 加 1。当 b 等于 1 时，GICR=0xE0 打开外部中断 0 和外部中断 1，即启动计数；当 b 等于 2 时，GICR=0x20，关闭外部中断 0 和外部中断 1，即暂停计数，延时 0.1s 后，再将 GICR=0xE0 恢复加 1/减 1 计数；当 b 等于 3 时，将 a 和 b 都清零，即计数清零，同时 GICR=0x20，关闭外部中断 0 和外部中断 1。

由于 LED 数码管采用动态扫描显示两位数据，因此对 a 需进行十位与个位的分离，将分离后的十位数字送给 data_H；分离后的个位数字送给 data_L。当 C1 有效时 P0=tab[data_H]；C2 有效时 P0=tab[data_L]，CS1、CS2 的有效切换较短，在 LED 数码管上即可动态显示 0～99。在此，绘制出外部中断函数 2 的程序流程如图 6-16 所示，其他函数的程序流程图可参考图 6-13。

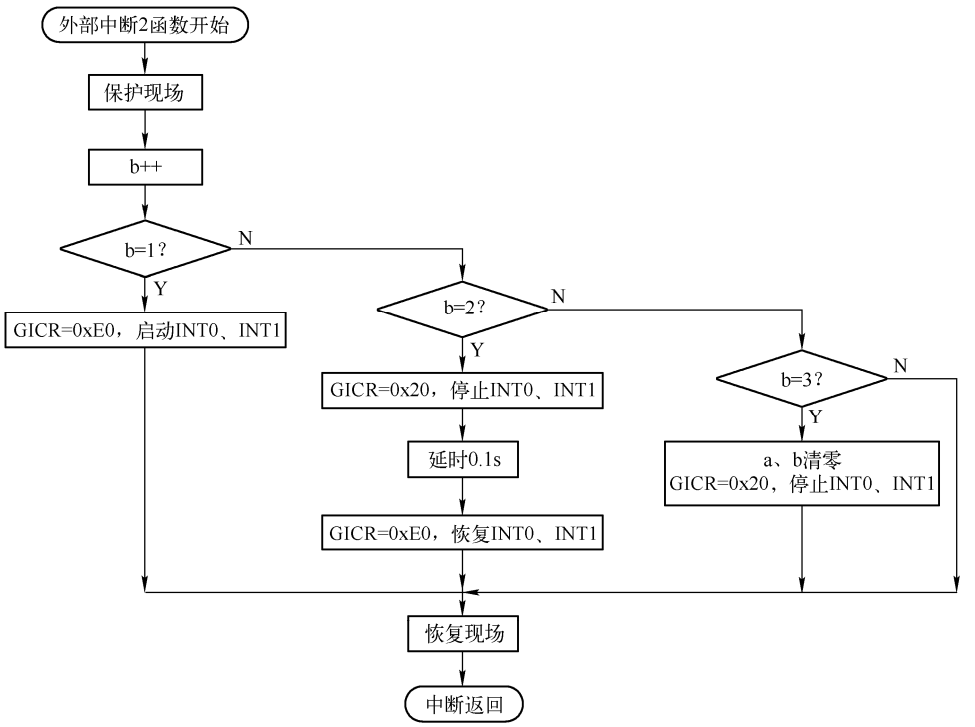


图 6-16　外部中断函数 2 的程序流程图

源程序

```
/*****************************************************
  文件名:          采用外中断实现计数和清零控制.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#define   uchar unsigned char
#define   uint unsigned int
#define   button1          PIND.2
#define   button2          PIND.3
#define   button3          PINB.2
```

```
#define   CS1              PORTB.0                    //CS1 片选
#define   CS2              PORTB.1                    //CS2 片选
#define   LED               PORTC
flash uchar tab[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,      //共阳极 LED0～F 的段码
                   0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E,0xBF,0x10};
uchar a;                                                  //计数值
uchar data_H,data_L;                                      //计数值高、低位
uchar b;                                                  //开始/暂停/清零选择
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
interrupt[EXT_INT0] void int0_isr(void)                   //INT0 中断函数
{
  delay(10);
  if(button1==0)                                          //再次确认按键，软件防抖
    {
      if(a==99)
        {
          a=0;
        }
      else
        {
          a++;
        }
    }
}
interrupt[EXT_INT1] void int1_isr(void)                   //INT1 中断函数
{
  delay(10);
  if(button2==0)                                          //再次确认按键，软件防抖
    {
      if(a==0)
        {
          a=99;
        }
      else
        {
          a--;
        }
    }
}
interrupt[EXT_INT2] void int2_isr(void)                   //INT2 中断函数
{
  delay(10);
```

```
    if(button3==0)                          //再次确认按键，软件防抖
      {
        b++;
        if(b==1)
          {                                 //允许计数
            GICR=0xE0;                      //允许 INT0、INT1、INT2 产生中断
          }
        if(b==2)                            //暂停计数
          {
            GICR=0x20;                      //只允许 INT2 产生中断
            delay(100);
            GICR=0xE0;
          }
        if(b>=3)                            //清零
          {
            b=0;
            a=0;
            GICR=0x20;                      //只允许 INT2 产生中断
          }
      }
}
void data_in(void)                          //获取高、低位
{
  data_L=a%10;
  data_H=a/10;
}
void display(void)
{
  LED=tab[data_H];
  CS1=1;
  delay(5);
  CS1=0;
  LED=tab[data_L];
  CS2=1;
  delay(5);
  CS2=0;
}
void port_init(void)                        //端口初始化
{
     DDRB=0xFB;                             //PB 端口配置为输出
    PORTB=0xFF;                             //PB 端口初始值设置
     DDRC=0xFF;                             //PC 端口配置为输出
    PORTC=0xFF;                             //PC 端口初始值设置
     DDRD=0xF3;                             //PD 端口配置为输出
    PORTD=0xFF;                             //PD 端口初始值设置
}
void INT_init(void)                         //中断初始化设置
{
```

```
    MCUCR=0x0A;                    //INT0、INT1 为下降沿时产生中断请求
   MCUCSR=0x00;                    //INT2 为下降沿时产生中断请求
     GICR=0x20;                    //允许 INT2 产生中断
     #asm("sei")                   //开启总中断
}
void main(void)
{
   port_init();
   INT_init();
   a=0;b=0;
   while(1)
     {
       data_in();
       display();
     }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“采用外中断进行计数和清零控制.COFF”文件，然后在 Proteus ISIS 中打开已创建的采用外中断进行计数和清零控制电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在按键 K1、K2、K3 都没有被按下的初始状态时，LED 数码管显示 00。如果按键 K3 没有被按下，不管按键 K1 或 K2 按下多少次，LED 数码管仍然显示为 00。第 1 次按下按键 K2 后，每次按下按键 K1，LED 数码管进行加 1 显示，当显示达 99 时，按键 K1 再次被按下，LED 数码管显示为 0；每次按下按键 K2，LED 数码管进行减 1 显示，当显示达 00 时，按键 K2 再次被按下，LED 数码管显示为 99。第 2 次按下按键 K2 时，暂停加 1/减 1 计数，按键 K1 或 K2 被按下后 LED 数码管显示暂停前的数值，当延时 0.1s 后，恢复加 1/减 1 计数。第 3 次按下按键 K2 时，LED 数码管显示为 00，并停止加 1/减 1 计数。第 4 次按下按键 K2 时，每次按下按键 K1，LED 数码管进行加 1 显示。

项目七　定时器/计数器控制的应用设计

〖知识目标〗

☺ 了解定时器/计数器的工作原理、定时器/计数器的控制寄存器定义、定时器/计数器的使用方法;

☺ 掌握定时器/计数器的控制及应用。

〖能力目标〗

☺ 学会定时器/计数器相关寄存器的设置;

☺ 会用中断的方式实现对单片机定时器/计数器的应用。

在单片机测量控制系统中，常需要有实时时钟的计数器，以实现定时（或延时）控制及对外界事件进行计数。ATmega16 单片机内部有 2 个 8 位和 1 个 16 位定时器/计数器，它们分别是 8 位定时器/计数器 0（T/C0）、定时器/计数器 2（T/C2）和 16 位定时器/计数器 1（T/C1）。本项目将介绍这 3 个定时器/计数器的使用方法。



任务 1　定时器/计数器控制原理认知

1．T/C0 的控制原理

1）T/C0 的工作原理　T/C0 是一个通用的带有输出比较匹配和 PWM 波形发生器的 8 位单通道定时器/计数器模块，它有两个 8 位的寄存器，即计数寄存器 TCNT0 和输出比较寄存器 OCR0。双缓冲的输出比较寄存器 OCR0 一直与 TCNT0 的数值进行比较，比较的结果可用于产生 PWM 波形，或者在输出比较引脚 OC0（PB3 端口）上产生变化频率的输出。

T/C0 可以选择来自外部引脚 T0（PB0 端口）的信号提供，也可以通过预分频器由内部时钟驱动。时钟控制模块控制使用哪个时钟源及哪个边沿来进行加或减计数。T/C0 的时钟分频器逻辑如图 7-1 所示，预分频器对系统分频后，作为 T/C0 的驱动时钟。T/C0 的时钟可以是系统时钟，或者系统时钟的 8 分频、64 分频、256 分频及 1024 分频，通过控制寄存器 TCCR0 的时钟选择位 CS02～CS00 控制。

当 T/C0 使用系统内部时钟作为计数源时，通常作为定时器和波形发生器使用。因为系统时钟的频率是已知的，所以通过计数器的计数值就可以知道时间值。当 T/C0 使用外部时钟作为计数源时，通常作为计数器使用，可用于记录外部脉冲的个数。

2）T/C0 的工作模式　T/C0 共有 4 种不同的工作模式，即普通模式、CTC 模式、快速 PWM 模式和相位修正 PWM 模式。这 4 种模式的选择及 OC0 不同方式的输出是由 T/C0 的控制寄存器 TCCR0 的标志位 WGM01、WGM00 和 COM01、COM00 的组合决定的。

【普通模式（WGM01、WGM00=00）】普通模式为最简单的工作模式。在此模式下，

计数器不停地累加。计到 8bit 的最大值后（TOP= 0xFF，即 TCNT0=0xFF），在下一个计数脉冲到来时便恢复为 0x00（TCNT0=0x00），并继续单向加 1 计数。在 TCNT0=0x00 的同一个定时器时钟里 T/C0 溢出标志 TOV0 置“1”。此时，TOV0 有点像第 9 位，只是只能置“1”，而不能清“0”。

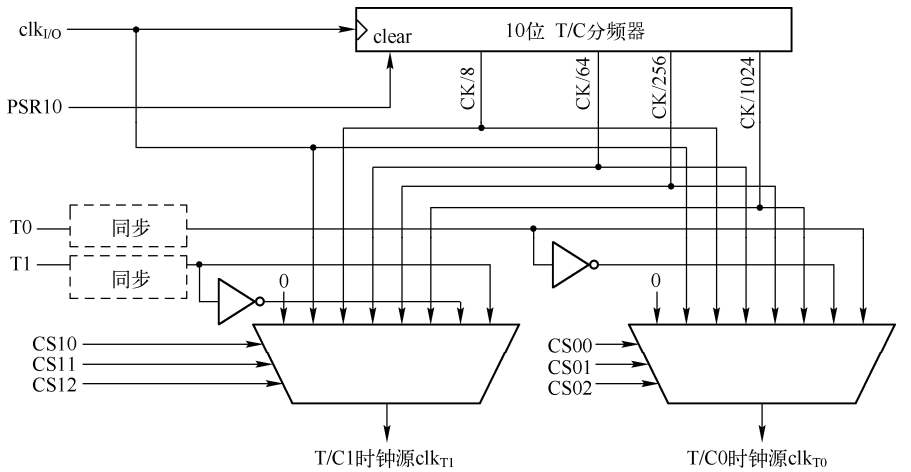


图 7-1　T/C0 的时钟预分频器逻辑图

T/C0 在普通模式下工作时，同样可以使用输出比较单元来产生定时中断。但是最好不要在普通模式下利用输出比较来产生波形，因为这会占用太多的 CPU 时间。

【CTC 模式（WGM01、WGM00=10）】CTC 模式即比较匹配时清零定时器模式。T/C0 工作在此模式下时，OCR0 寄存器用于调节计数器的分辨率。当计数器的数值 TCNT0 等于 OCR0 时，计数器清零。OCR0 定义了计数器的 TOP 值，即计数器的分辨率。这个模式使得用户可以很容易地控制比较匹配输出的频率，也简化了外部事件计数的操作。

利用 OCF0 标志可以在计数器数值达到 TOP 时产生中断。在中断服务程序中可以更新 TOP 的数值。由于 CTC 模式没有双缓冲功能，在计数器中没有使用预分频器或使用很低的预分频器工作时，将 TOP 更改为接近 BOTTOM 的数值时要小心。如果写入的 OCR0 数值小于当前 TCNT0 的数值，计数器将丢失一次比较匹配。在下一次比较匹配发生前，计数器不得不先计数到最大值 0xFF，然后再从 0x00 开始计数到 0xOCF0。

为了在 CTC 模式下得到波形输出，可以设置 OC0 在每次比较匹配发生时改变逻辑电平。这可以通过设置 COM01、COM00=01 来完成。在期望获得 OC0 输出前，首先要将其端口设置为输出。波形发生器能够产生的最大频率为 $f_{OC0}=f_{clk_{I/O}}/2$（OCR0=0x00）。频率由如下公式确定:

$$f_{OC0}=\frac{f_{clk_{I/O}}}{2N(1+OCR0)}$$

式中，变量 N 代表预分频因子（1、8、64、256 或 1024），下同。

【快速 PWM 模式（WGM01、WGM00=11）】PWM 即脉冲宽度调制（Pulse Width Modulation）。T/C0 工作在快速 PWM 模式下时，可用于产生高频的 PWM 波形。快速 PWM 模式与其他 PWM 模式的不同之处是其单斜波工作方式。计数器从 BOTTOM 计数到 TOP，然后立即回到 BOTTOM 重新开始。

工作于快速 1PWM 模式时，计数器的数值一直增加到 TOP，然后在后面的一个时钟周期清零。计时器数值达到 TOP 时，T/C0 溢出标志 TOV0 置位。如果中断使能，在中断服务程序可以更新比较值。工作于快速 PWM 模式时，比较单元可以在 OC0 引脚上输出 PWM 波形。设置 COM01、COM00=10 可以产生普通的 PWM 信号；COM01、COM00=11，则可以产生反向 PWM 波形。要想在引脚上得到输出信号，还必须将 OC0 的数据方向设置为输出。OC0 输出的 PWM 波形的频率由如下公式确定:

$$f_{\mathrm{OC0PWM}}=\frac{f_{\mathrm{clk_{I/O}}}}{256N}$$

【相位修正 PWM 模式（WGM01、WGM00=01）】相位修正 PWM 模式（WGM01、WGM00=01）为用户提供了一个获得高精度相位修正 PWM 波形的方法。此模式基于双斜波操作。计时器重复地从 BOTTOM（即 0x00）计数到 TOP（即 0xFF），然后又从 TOP 倒退回到 BOTTOM。在一般的比较输出模式下，当计时器往 TOP 计数时，若发生了 TCNT0 与 OCR0 的匹配，OC0 将清零为低电平；而在计时器往 BOTTOM 计数时，若发生了 TCNT0 与 OCR0 的匹配，OC0 将置位为高电平。工作于反向输出比较时，则正好相反。与单斜波操作相比，双斜波操作可获得的最大频率要小。但由于其对称的特性，十分适合于电机控制。

工作于相位修正 PWM 模式时，比较单元可以在 OC0 引脚产生 PWM 波形。设 COM01、COM00=10 时，产生普通相位的 PWM；COM01、COM00=11 时，产生反向 PWM 信号。若想在引脚上得到输出信号，还必须将 OC0 的数据方向设置为输出。OCR0 和 TCNT0 比较匹配发生时，OC0 寄存器将产生相应的清零或置位操作，从而产生 PWM 波形。工作于相位修正模式时，PWM 频率可由如下公式确定:

$$f_{\mathrm{OC0PCPWM}}=\frac{f_{\mathrm{clk_{I/O}}}}{510N}$$

3）T/C0 的相关寄存器　T/C0 有两个 8 位的寄存器，即计数寄存器 TCNT0 和输出比较寄存器 OCR0。其他相关的寄存器还有 T/C0 的控制寄存器 TCCR0、中断标志寄存器 TIFR 和定时器中断屏蔽寄存器 TIMSK。

【T/C0 控制寄存器 TCCR0】其各位定义如下:

Bit	7	6	5	4	3	2	1	0
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00
读/写	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ FOC0（bit7）: 强制输出比较位。FOC0 位只在 T/C0 被设置为非 PWM 模式下工作时才有效，但为了与未来器件的兼容性，在使用 PWM 时，写 TCCR0 时要对其清零。对其写“1”后，波形比较器将立即进行比较操作，比较匹配输出引脚 OC0 将按照 COM01 和 COM00 设置输出相应的电平。

> 注意:
> FOC0 类似于一个选通脉冲，而 OC0 的输出还取决于 COM01 和 COM00 位的设置。

一个 FOC0 选通脉冲不会产生任何中断申请，也不影响计数器 TCNT0 的寄存器 OCR0 的值。一旦一个真正的比较匹配发生，则 OC0 的输出将根据 COM01 和 COM00 位

的设置而更新。

☺ WGM01、WGM00（bit3、bit6）：波形发生模式位。这两个标志位控制 T/C0 的计数和工作方式、计数器计数上限值及确定波形发生器的工作模式。T/C0 支持的有普通模式、CTC 模式及两种 PWM 模式，见表 7-1。

表 7-1　T/C0 的波形产生模式

模式	WGM01	WGM00	T/C0 的工作模式	最大值	OCR0 的更新时间	TOV0 的置位时刻
0	0	0	普通模式	0xFF	立即更新	0xFF（TOP）
1	0	1	相位修正的 PWM 模式	0xFF	0xFF（TOP）	0x00（BOTTOM）
2	1	0	CTC 模式	OCR0	立即更新	0xFF（TOP）
3	1	1	快速 PWM 模式	0xFF	0xFF（TOP）	0xFF（TOP）

☺ COM01、COM00（bit5、bit4）：波形模式产生位。这两个位用于控制比较输出引脚 OC0（PB3）的输出方式。若 COM01、COM00 中的任何一位或两位置“1”，则 OC0 的输出将覆盖 PB3 引脚的通用 I/O 端口功能，而以比较匹配输出的方式进行工作，但此时其方向控制位（DDRB.3）需要设置为 1（输出方式）。当引脚 PB3 作为 OC0 输出引脚时，其输出方式取决于 COM01、COM00 和 WGM01、WGM00 的设定，见表 7-2。

表 7-2　COM01、COM00 位功能定义

工 作 模 式	COM01	COM00	说　　明	备　　注
0 或 2（普通模式或 CTC 模式，即非 PWM 模式）	0	0	正常的端口操作，不与 OC0 引脚物理连接	WGM01、WGM00=00 或 10
	0	1	比较匹配发生时，触发 OC0（OC0 为原 OC0 的取反）	
	1	0	比较匹配发生时，OC0 清零	
	1	1	比较匹配发生时，OC0 置位	
3（快速 PWM 模式）	0	0	PB3 为通用 I/O 引脚（OC0 与引脚不连接）	WGM01、WGM00=11
	0	1	保留	
	1	0	比较匹配时，清零 OC0；计数值为 0xFF 时，置位 OC0	
	1	1	比较匹配时，置位 OC0；计数值为 0xFF 时，清零 OC0	
1（相位修正的 PWM 模式）	0	0	PB3 为通用 I/O 引脚（OC0 与引脚不连接）	WGM01、WGM00=01
	0	1	保留	
	1	0	向上计数过程中比较匹配时，清零 OC0 向下计数过程中比较匹配时，置位 OC0	
	1	1	向上计数过程中比较匹配时，置位 OC0 向下计数过程中比较匹配时，清零 OC0	

☺ CS02、CS01、CS00（bit2~bit0）：时钟选择位。用于设定 T/C0 的时钟源，见表 7-3。

表 7-3　T/C0 时钟源的设定

CS02	CS01	CS00	说　　明
0	0	0	无时钟，T/C0 停止工作

续表

CS02	CS01	CS00	说　明
0	0	1	$clk_{I/O}$/1（没有预分频）
0	1	0	$clk_{I/O}$/8（来自预分频）
0	1	1	$clk_{I/O}$/64（来自预分频）
1	0	0	$clk_{I/O}$/256（来自预分频）
1	0	1	$clk_{I/O}$/1024（来自预分频）
1	1	0	时钟由 T0 引脚引入，下降沿触发
1	1	1	时钟由 T0 引脚引入，上升沿触发

【T/C0 计数寄存器 TCNT0】通过 T/C0 寄存器，MCU 可以直接对计数器的 8 位数据进行读/写访问。写 TCNT0 寄存器将在下一个定时器时钟周期中阻塞比较匹配。因此，在计数器运行期间修改 TCNT0 的内容，有可能丢失一次 TCNT0 与 OCR0 的匹配比较操作。其各位定义如下：

Bit	7	6	5	4	3	2	1	0
	TCNT0[7..0]							
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

【输出比较寄存器 OCR0】输出比较器包含一个 8 位的数据，用于与寄存器 TCNT0 中的计数值进行匹配比较。在 T/C0 运行期间，比较匹配单元一直将寄存器 TCNT0 的计数值与寄存器 OCR0 的内容进行比较。一旦 TCNT0 的计数值与 OCR0 的数据匹配相等，将产生一个输出比较匹配相等的中断申请，或者在 OC0 引脚上产生波形。其各位定义如下：

Bit	7	6	5	4	3	2	1	0
	OCR0[7..0]							
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

【定时器/计数器中断屏蔽寄存器 TIMSK】其各位定义如下：

Bit	7	6	5	4	3	2	1	0
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ OCIE0（bit1）：T/C0 输出比较匹配中断使能位。当 OCIE0 和状态寄存器 SREG 的全局中断使能使位 I 置位时，T/C0 的输出比较匹配中断使能。当 T/C0 的比较匹配事件发生，即 TIFR 中的 OCF0 置位时，产生输出比较匹配中断。

☺ TOIE0（bit0）：T/C0 溢出中断使能位。当 TOIE0 和状态寄存器 SREG 的全局中断使能位 I 都置位时，T/C0 的溢出中断使能。当 T/C0 发生计数溢出，即 TIFR 中的 TOV0 置位时，产生溢出中断。

【中断标志寄存器 TIFR】其各位定义如下：

Bit	7	6	5	4	3	2	1	0
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ OCF0（bit1）：T/C0 输出比较匹配中断标志位。当 T/C0 与 OCR0（输出比较寄存器）的值匹配时，OCF0 置位（无论中断是否使能）。此位在执行中断服务程序时硬件清零，也可以由软件写“1”清零。当 SREG 寄存器中的 I 和 TIMSK 中的 OCIE0 及 OCF0 都置位时，中断服务程序得到执行。

☺ TOV0（bit0）：T/C0 溢出中断标志位。当 T/C0 计数溢出时，TOV0 置位（无论中断是否使能）。此位在执行中断 1 服务程序时，硬件清零，也可以由软件写“1”清零。当 SREG 寄存器中的 I 位和 TIMSK 中的 TOIE0 都置位时，中断服务程序得到执行。

📖 注意：
在相位修正的 PWM 模式中，当 T/C0 在 0x00 改变计数方向时 TOV0 置位。

2．T/C1 的控制原理

1）T/C1 的工作原理 T/C1 是一个通用的带有一路噪声抑制输入捕获、两路独立输出比较的 PWM 波形发生器的 16 位单通道定时器/计数器模块。T/C1 的计数时钟源和 T/C0 的计数时钟源类似，可以来自于内部，也可以来自于外部。T/C1 与 T/C0 共用一个分频器，如图 7-1 所示。分频器对系统时钟分频后作为 T/C1 的驱动时钟。时钟源由位于 T/C1 控制寄存器 B（TCCR1B）的时钟选择位 CS12～CS10 决定。

16 位计数器映射到两个 8 位 I/O 存储器的位置中，TCNT1H 为高 8 位，TCNT1L 为低 8 位。CPU 只能间接访问 TCNT1H 寄存器。CPU 访问 TCNT1H 时，实际访问的是临时寄存器（TEMP）。读取 TCNT1L 时，临时寄存器的内容更新为 TCNT1H 的数值；而对 TCNT1L 执行写操作时，TCNT1H 被临时寄存器的内容所更新。

T/C1 的输入捕捉单元可用于捕获外部事件，并对其进行时间标志以说明此时间的发生时刻。外部事件发生的触发信号由引脚 ICP1（PD6）输入，也可通过模拟比较器单元来产生。时间标志可用于计算频率、占空比及信号的其他特征，以及为事件创建日志。当引脚 ICP1 上的逻辑电平（事件）发生了变化，或者模拟比较器输出 ACO 电平发生了变化，并且这个电平变化为边沿检测器所证实，输入捕捉即被激发，16 位的 TCNT1 数据被复制到输入捕捉寄存器 ICR1，同时输入捕捉标志位 ICF1 置位。如果此时 ICIE1=1，输入捕捉标志将产生输入捕捉中断。中断执行时 ICF1 自动清零，或者也可通过软件在其对应的 I/O 位置写入逻辑“1”清零。

16 位比较器持续比较 TCNT1 与 OCR1x 的内容，一旦发现它们相等，比较器立即产生一个匹配信号，然后 OCF1x 在下一个定时器时钟置位。如果此时 OCIE1x=1，OCF1x 置位，将引发输出比较中断。中断执行时，OCF1x 标志自动清零，或者通过软件在其相应的 I/O 位置写入逻辑“1”也可以清零。根据 WGM13～WGM10 与 COM1x1～COM1x0 的不同设置，波形发生器用匹配信号生成不同的波形。

2）T/C1 的工作模式 T/C1 有 5 种不同的工作模式，即普通模式、CTC 模式、快速 PWM 模式和相位修正模式。到底使用哪种模式是由波形发生器模式（WGM13～WGM10）和比较输出模式（COM1x1、COM1x0）的控制位决定的。

【普通模式（WGM13～WGM10=0000）】普通模式为最简单的工作模式。在此模式下计数器不停地累加。计到 16bit 的最大值后（TOP=0xFFFF，TCNT1=0xFFFF，即 TCNT1H=0xFF、TCNT1L=0xFF），在下一个计数脉冲到来时 TCNT1 便恢复为 0x0000（TCNT1H=

0x00、TCNT1L=0x00），并继续单向加 1 计数。在 TCNT1=0x0000 的同一个定时器时钟中 T/C1 溢出标志 TOV1 置“1”。此时 TOV1 有点像第 17 位，只是只能置“1”，而不能清“0”。但由于定时器中断服务程序能够自动清零 TOV1。

【CTC 模式（WGM13～WGM10=0100 或 1100）】T/C1 在 CTC 模式下，OCR1A 或 ICR1 寄存器用于调节计数器的分辨率。当计数器的数值 TCNT1 等于 OCR1A（WGM13～WGM10=0100）或 ICR1（WGM13~WGM10=1100）时，计数器清零。OCR1A 或 ICR1 定义了计数器的 TOP 值，即计数器的分辨率。

利用 OCF1A 或 ICF1 标志可以在计数器数值达到 TOP 时产生中断。在中断服务程序中可以更新 TOP 的数值。由于 CTC 模式没有双缓冲功能，在计数器以无预分频器或很低的预分频器工作时，将 TOP 更改为接近 BOTTOM 的数值时要小心，如果写入的 OCR1A 或 ICR1 的数值小于当前 TCNT1 的数值，计数器将丢失一次比较匹配。在下一次比较匹配发生前，计数器不得不先计数到最大值 0xFFFF，然后再从 0x0000 开始计数到 OCR1A 或 ICR1。

为了在 CTC 模式下得到波形输出，可以设置 OC1A 在每次比较匹配发生时改变逻辑电平，这可以通过设置 COM1A1、COM1A0=01 来完成。在期望获得 OC1A 输出前，首先要将其端口设置为输出（DDR_OC1A=1）。波形发生器能够产生的最大频率为 $f_{\mathrm{OC1A}}=f_{\mathrm{clk_{I/O}}}/2$（OCR1A= 0x0000）。频率由如下公式确定：

$$f_{\mathrm{OC1A}}=\frac{f_{\mathrm{clk_{I/O}}}}{2N(1+\mathrm{OCR1}A)}$$

【快速 PWM 模式（WGM13～WGM10=0101、0110、0111、1110 或 1111）】T/C1 工作在快速 PWM 模式下时，也可用于产生高频的 PWM 波形。快速 PWM 模式与其他 PWM 模式的不同之处在于其单边斜边工作方式。计数器从 BOTTOM 计数到 TOP，然后立即回到 BOTTOM 重新开始。对于普通的比较输出模式，输出比较引脚 OC1x 在 TCNT1 与 OCR1x 匹配时置位，在 TOP 时清零；对于反向比较输出模式，OCR1x 的动作正好相反。由于使用了单边斜坡模式，快速 PWM 模式的工作频率比使用双斜波的相位修正 PWM 模式高 1 倍。此高频操作特性使得快速 PWM 模式十分适合于功率调节、整流和 DAC 应用。

工作于快速 PWM 模式时，PWM 分辨率可固定为 8、9 或 10 位，也可由 ICR1 或 OCR1A 定义。最小分辨率为 2bit（ICR1 或 OCR1A 设为 0x0003），最大分辨率为 16 位（ICR1 或 OCR1A 设为 TOP）。PWM 分辨率位数可用下式计算：

$$R_{\mathrm{FPWM}}=\frac{\ln(\mathrm{TOP}+1)}{\ln(2)}$$

工作于快速 PWM 模式时，比较单元可以在 OC1x 引脚上输出 PWM 波形。设置 COM1x1、COM1x0=10，可以产生普通的 PWM 信号；设置 COM1x1、COM1x0=11，则可以产生反向 PWM 波形。此外，要真正从物理引脚上输出信号，还必须将 OC1x 的数据方向 DDR_OC1x 设置为输出。输出的 PWM 波形的频率由下式确定：

$$f_{\mathrm{OC1PWM}}=\frac{f_{\mathrm{clk_{I/O}}}}{N(1+\mathrm{TOP})}$$

【相位修正 PWM 模式（WGM13～WGM10=0001、0010、0011、1010、1011）】与相位和频率修正模式类似，此模式基于双斜波操作。计时器重复地从 BOTTOM 计数到 TOP，然后又从 TOP 倒退回到 BOTTOM。在一般的比较输出模式下，当计时器往 TO 计数时，若

TCNT1 与 OCR1x 匹配，OC1x 将清零为低电平；而在计时器往 BOTTOM 计数时，若 TCNT1 与 OCR1x 匹配，OC1x 将置位为高电平。工作于反向比较输出时则正好相反。与单斜波操作相比，双斜波操作可获得的最大频率比它要小，但其对称特性十分适合于电机控制。

相位修正 PWM 模式的 PWM 分辨率固定为 8、9 或 10 位，也可以由 ICR1 或 OCR1A 定义。最小分辨率为 2bit（ICR1 或 OCR1A 设为 0x0003），最大分辨率为 16bit（ICR1 或 OCR1A 设为 TOP）。PWM 分辨率位数可用下式计算:

$$R_{\mathrm{PCWM}}=\frac{\ln(\mathrm{TOP}+1)}{\ln(2)}$$

工作于相位修正 PWM 模式时，比较单元可以在 OC1x 引脚输出 PWM 波形。设置 COM1x1、COM1x0=10，可以产生普通的 PWM；设置 COM1x1、COM1x0=11 可以产生反向 PWM。要真正从物理引脚上输出信号，还必须将 OC1x 的数据方向 DDR_OC1x 设置为输出。OCR1x 和 TCNT1 比较匹配发生时，OC1x 寄存器将产生相应的清零或置位操作，从而产生 PWM 波形。工作于相位修正模式时，PWM 频率可由下式确定:

$$f_{\mathrm{OC1xPCPWM}}=\frac{f_{\mathrm{clk_{I/O}}}}{2\cdot N\cdot \mathrm{TOP}}$$

【相位与频率修正 PWM 模式（WGM13～WGM10=1000 或 1001）】T/C1 工作在相位与频率修正 PWM 模式下时，可以产生高精度的、相位与频率都准确的 PWM 波形。与相位修正模式类似，相位与频率修正 PWM 模式基于双斜波操作。定时器重复地从 BOTTOM 计数到 TOP，然后又从 TOP 倒退回到 BOTTOM。在一般的比较输出模式下，当定时器往 TOP 计数时，若 TCNT1 与 OCR1x 匹配，OC1x 将清零为低电平；而在定时器往 BOTTOM 计数时，若 TCNT1 与 OCR1x 匹配，OC1x 将置位为高电平。工作于反向输出比较时则正好相反。与单斜波的操作相比，双斜波操作可获得的最大频率要小于单斜波，但其对称特性十分适合于电机控制。

相位与频率修正 PWM 模式的 PWM 分辨率可由 ICR1 或 OCR1A 定义。最小分辨率为 2bit（CR1 或 OCR1A 设为 0x0003），最大分辨率为 16bit（ICR1 或 OCR1A 设为 TOP）。PWM 分辨率位数可用下式计算:

$$R_{\mathrm{PFCPWM}}=\frac{\ln(\mathrm{TOP}+1)}{\ln(2)}$$

工作于相位与频率修正 PWM 模式时，比较单元可以在 OC1x 引脚上输出 PWM 波形。设置 COM1x1、COM1x0=10，可以产生普通的 PWM 信号；COM1x1、COM1x0=11，可以产生反向 PWM 波形。要想真正输出信号，还必须将 OC1x 的数据方向设置为输出。产生 PWM 波形的原理是，OC1x 寄存器在 OCR1x 与升序计数的 TCNT1 匹配时置位（或清零），与降序计数的 TCNT1 匹配时清零（或置位），从而产生 PWM 波形。输出的 PWM 频率可以通过下式计算得到:

$$f_{\mathrm{OC1xPFCPWM}}=\frac{f_{\mathrm{clk_{I/O}}}}{2\cdot N\cdot \mathrm{TOP}}$$

3）T/C1 的相关寄存器

【T/C1 控制寄存器 TCCR1A】其各位定义如下:

Bit	7	6	5	4	3	2	1	0
	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10
读/写	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ COM1A1、COM1A0（bit7、bit6）：通道 A 的比较输出模式。

☺ COM1B1、COM1B0（bit5、bit4）：通道 B 的比较输出模式。

COM1A1、COM1A0 与 COM1B1、COM1B0 分别控制 OC1A 与 OC1B 状态。若 COM1A1、COM1A0（COM1B1、COM1B0）的一位或两位写入“1”时，OC1A（OC1B）输出功能将取代 I/O 端口功能。此时 OC1A（OC1B）相应的输出引脚数据方向控制必须置“1”。OC1A（OC1B）与物理引脚相连时，COM1x1~COM1x0（x 表示 A/B）的功能由 WGM13~WGM10 的设置决定，见表 7-4。

表 7-4　比较输出模式功能定义

工作模式	COM1A1(COM1B1)	COM1A0(COM1B0)	说　明
普通模式	0	0	普通端口操作，非 OC1A/OC1B 功能
	0	1	比较匹配时，OC1A/OC1B 电平取反
	1	0	比较匹配时，清零 OC1A/OC1B（输出低电平）
	1	1	比较匹配时，置位 OC1A/OC1B（输出高电平）
快速 PWM 模式	0	0	普通端口操作，非 OC1A/OC1B 功能
	0	1	WGM13～WGM10=1111 时，比较匹配 OC1A 取反，OC1B 不占用物理引脚；WGM13～WGM10 为其他值时，为普通端口操作，非 OC1A/OC1B 功能
	1	0	比较匹配时，清零 OC1A/OC1B，OC1A/OC1B 在 TOP 时置位
	1	1	比较匹配时，置位 OC1A/OC1B，OC1A/OC1B 在 TOP 时清零
相位修正 PWM 模式	0	0	普通端口操作，非 OC1A/OC1B 功能
	0	1	WGM13～WGM10=1001 时，比较匹配 OC1A 取反，OC1B 不占用物理引脚；WGM13～WGM10 为其他值时，为普通端口操作，非 OC1A/OC1B 功能
	1	0	升序计数时，比较匹配将清零 OC1A/OC1B；降序计数时，比较匹配将置位 OC1A/OC1B
	1	1	升序计数时，比较匹配将置位 OC1A/OC1B；降序计数时，比较匹配将清零 OC1A/OC1B

☺ FOC1A、FOC1B（bit3、bit2）：通道 A/B 强制输出比较。FOC1A/FOC1B 只有当 WGM13~WGM10 指定为非 PWM 模式时被激活。为了与未来器件兼容，当 T/C1 工作在 PWM 模式下对 TCCR1A 写入时，这两位必须清零。当 FOC1A/FOC1B 位置“1”时，立即强制波形产生单元进行比较匹配。COM1x1、COM1x:0 的设置改变 OC1A/OC1B 的输出。

🕮 注意:

FOC1A/FOC1B 位作为选通信号，COM1x1、COM1x0 位的值决定强制比较的效果。

在 CTC 模式下使用 OCR1A 作为 TOP 值，FOC1A/FOC1B 选通既不会产生中断也不会清除定时器。FOC1A/FOC1B 位读出的数据总是为“0”。

☺ WGM11、WGM10（bit1、bit0）：波形发生模式。这两位与位于 TCCR1B 寄存器的 WGM13、WGM12 相结合，用于控制计数器计数的上限值和确定波形发生器的工作模式。T/C1 支持的工作模式分为普通模式（计数器）、比较匹配时清零定时器（CTC）模式，以及 3 种脉宽调制（PWM）模式，见表 7-5。

表 7-5　波形发生模式

模式	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	定时器/计数器工作模式	计数上限值 TOP	OCR1x 更新时刻	TOV1 置位时刻
0	0	0	0	0	普通模式	0xFFFF	立即更新	TOP
1	0	0	0	1	8 位相位修正 PWM	0x00FF	TOP	BOTTOM
2	0	0	1	0	9 位相位修正 PWM	0x01FF	TOP	BOTTOM
3	0	0	1	1	10 位相位修正 PWM	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	立即更新	TOP
5	0	1	0	1	8 位快速 PWM	0x00FF	TOP	BOTTOM
6	0	1	1	0	9 位快速 PWM	0x01FF	TOP	BOTTOM
7	0	1	1	1	10 位快速 PWM	0x03FF	TOP	BOTTOM
8	1	0	0	0	相位与频率修正 PWM	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	相位与频率修正 PWM	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	相位修正 PWM	ICR1	TOP	BOTTOM
11	1	0	1	1	相位修正 PWM	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	立即更新	TOP
13	1	1	0	1	保留	—	—	—
14	1	1	1	0	快速 PWM	ICR1	TOP	TOP
15	1	1	1	1	快速 PWM	OCR1A	TOP	TOP

【T/C1 控制寄存器 TCCR1B】其各位定义如下：

Bit	7	6	5	4	3	2	1	0
	ICNC1	ICES1	—	WGM13	WGM12	CS12	CS11	CS10
读/写	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ ICNC1（bit7）：输入捕获噪声抑制使能位。ICNC1=1 时，允许输入捕获噪声抑制功能，此时外部引脚 ICP1（PD6）的输入被滤波，其作用是从 ICP1 引脚连续进行 4 次采样。如果 4 次采样值都相同，则信号将送入边沿检测器。所以，允许该功能使得输入捕获被延迟了 4 个时钟周期。

☺ ICES1（bit6）：输入捕获触发边沿选择。ICES1=1 时，选择逻辑电平的上升沿触发输入捕获；ICES1=0 时，选择下降沿触发输入捕获。

☺ bit5：保留位，为与将来的器件相兼容，写 TCCR1B 时，此位必须写入“0”。

☺ WGM13、WGM12（bit4、bit3）：波形发生模式。这两位与位于 TCCR1A 寄存器的 WGM11、WGM10 相结合，用于控制计数器计数的上限值和确定波形发生器的工作模式。

☺ CS12、CS11、CS10（bit2、bit1、bit0）：T/C1 时钟源选择位。用于设定 T/C1 的时钟

源，见表 7-6。

表 7-6　T/C1 时钟源的设定

CS12	CS11	CS10	说　明
0	0	0	无时钟，T/C1 停止工作
0	0	1	$clk_{I/O}$/1（没有预分频）
0	1	0	$clk_{I/O}$/8（来自预分频）
0	1	1	$clk_{I/O}$/64（来自预分频）
1	0	0	$clk_{I/O}$/256（来自预分频）
1	0	1	$clk_{I/O}$/1024（来自预分频）
1	1	0	时钟由 T1 引脚引入，下降沿触发
1	1	1	时钟由 T1 引脚引入，上升沿触发

【T/C1 计数寄存器 TCNT1】在计数器运行期间，修改 TCNT1 的内容有可能丢失一次 TCNT1 与 OCR1x 的比较匹配操作。写 TCNT1 寄存器将在下一个定时器周期阻塞比较匹配。其各位定义如下：

Bit	7	6	5	4	3	2	1	0	
	TCNT1[15..8]								TCNT1H
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始值	0	0	0	0	0	0	0	0	
Bit	7	6	5	4	3	2	1	0	
	TCNT1[7..0]								TCNT1L
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始值	0	0	0	0	0	0	0	0	

【T/C1 输出捕获寄存器 OCR1A、OCR1B】其各位定义如下：

Bit	7	6	5	4	3	2	1	0	
	OCR1A[15..8]								OCR1AH
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始值	0	0	0	0	0	0	0	0	
Bit	7	6	5	4	3	2	1	0	
	OCR1A[7..8]								OCR1AL
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始值	0	0	0	0	0	0	0	0	
Bit	7	6	5	4	3	2	1	0	
	OCR1B[15..8]								OCR1BH
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始值	0	0	0	0	0	0	0	0	
Bit	7	6	5	4	3	2	1	0	
	OCR1B[7..0]								OCR1BL
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始值	0	0	0	0	0	0	0	0	

【T/C1 输入捕获寄存器 ICR1】ICR1 为 16 位的输入捕获寄存器，由两个 8 位寄存器 ICR1H 和 ICR1L 组成。当外部引脚 ICP1（或 T/C1 的模拟比较器）有输入捕获信号产生

时，计数器 TCNT1 中的数据将自动写入 ICR1 中。ICR1 的设定值可作为计数器的最大计数值。其各位定义如下：

Bit	7	6	5	4	3	2	1	0	
	ICR1[15..8]								ICR1H
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始值	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
	ICR1[7..0]								ICR1L
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始值	0	0	0	0	0	0	0	0	

【T/C1 中断屏蔽寄存器 TIMSK】其各位定义如下：

Bit	7	6	5	4	3	2	1	0
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ TICIE1（bit5）：T/C1 输入捕获中断使能位。当 TICIE1 和状态寄存器 SREG 的全局中断使能位 I 都置位时，T/C1 的输入捕获中断使能。一旦 TIFR 的 ICF1 置位，CPU 即开始执行 T/C1 输入捕获中断服务程序。

☺ OCIE1A（bit4）：T/C1 输出比较匹配 A 中断使能位。当 OCIE1A 和状态寄存器 SREG 的全局中断使能位 I 都置位时，T/C1 的输出比较匹配中断使能。一旦 TIFR 的 OCF1A 置位，CPU 即开始执行 T/C1 输出比较匹配 A 中断服务程序。

☺ OCIE1B（bit3）：T/C1 输出比较匹配 B 中断使能位。当 OCIE1B 和状态寄存器 SREG 的全局中断使能位 I 都置位时，T/C1 的输出比较匹配中断使能。一旦 TIFR 的 OCF1B 置位，CPU 即开始执行 T/C1 输出比较匹配 B 中断服务程序。

☺ TOIE1（bit2）：T/C0 溢出中断使能位。当 TOIE1 和状态寄存器 SREG 的全局中断使能位 I 都置位时，T/C1 的溢出中断使能。当 T/C1 发生计数溢出，即 TIFR 中的 TOV1 置位时，产生溢出中断。

【T/C1 中断标志寄存器 TIFR】其各位定义如下：

Bit	7	6	5	4	3	2	1	0
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ ICF1（bit5）：T/C1 输入捕获中断标志位。不管 TICIE1 是否为“1”，只要外部引脚 ICP1 出现捕获事件时，ICF1 将置“1”。此外，当 ICR1 作为计数器的 TOP 值时，一旦计数器值达到 TOP，ICF1 也将置“1”。执行输入捕捉中断服务程序时，ICF1 自动清零，用户也可以对其写入逻辑“1”来清除该标志位。

☺ OCF1A（bit4）：T/C1 输出比较 A 匹配标志位。当 TCNT1 与 OCR1A 匹配成功时，该位被设置为“1”。强制输出比较（FOC1A）不会置位 OCF1A。执行强制输出比较匹配 A 中断服务程序时，OCF1A 自动清零。也可以对其写入逻辑“1”来清除该标志位。

☺OCF1B（bit3）：T/C1 输出比较 B 匹配标志位。当 TCNT1 与 OCR1B 匹配成功时，该位被设置为“1”。强制输出比较（FOC1B）不会置位 OCF1B。执行强制输出比较匹配 B 中

断服务程序时，OCF1B 自动清零。也可以对其写入逻辑“1”来清除该标志位。

☺ TOV1（bit2）：T/C1 溢出标志。工作于普通模式和 CTC 模式时，一旦 T/C1 溢出，TOV1 将置“1”。工作在其他模式下，TOV1 标志位置“1”的时刻请参见表 7-5。执行溢出中断服务程序时，TOV1 自动清零。用户也可以对其写入逻辑“1”来清除该标志位。

【特殊功能 I/O 寄存器 SFIOR】 其各位定义如下：

Bit	7	6	5	4	3	2	1	0
	ADTS2	ADTS1	ADTS0	—	ACME	PUD	PSR2	PSR10
读/写	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ PSR10（bit0）：T/C1 与 T/C0 预分频器复位位。PSR10=1 时，T/C1 与 T/C0 预分频器复位，操作完成后这一位由硬件清零。PSR10=0 时，不会引起任何操作。T/C1 与 T/C0 共用这一预分频器，且预分频复位时对两个定时器都有影响。读该位总是返回 0。

3．T/C2 的控制原理

1）T/C2 的工作原理 T/C2 是一个通用的单通道带有输出比较匹配的 PWM 波形发生器的 8 位定时器/计数器模块。T/C2 可以由通过预分频器的内部时钟驱动，或者通过 TOSC1/2 引脚（PC6、PC7）接入的异步时钟驱动。

T/C2 的时钟预分频器逻辑如图 7-2 所示，时钟选择逻辑模块的输出称为 clk_{T2S}。在默认情况下，clk_{T2S} 与系统主时钟 $clk_{I/O}$ 连接。若置位 ASSR 的 AS2，T/C2 将由引脚 TOSC1 异步驱动，使得 T/C2 可以用做一个实时时钟 RTC。此时 TOSC1 和 TOSC2 从端口 C 脱离，引脚上可外接一个时钟晶振。

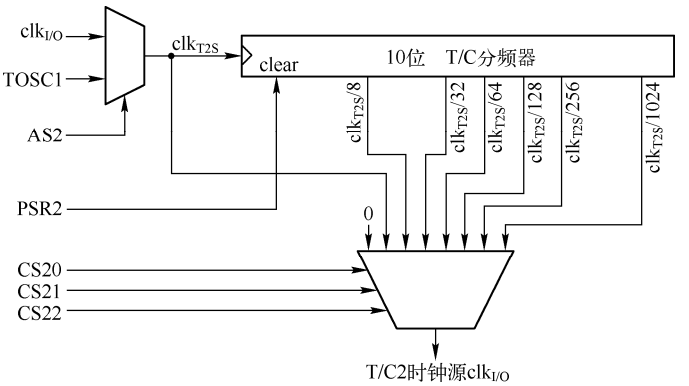


图 7-2　T/C2 的时钟预分频逻辑图

T/C2 的计数单元是一个可编程的 8 位双向计数器，计数器针对每一个 clk_{T2S} 实现清零、加一或减一操作。clk_{T2S} 可以由内部时钟源或外部时钟源产生，具体由时钟选择位 CS22～CS20 确定。没有选择时钟源（CS22～CS20=000）时，定时器停止。

计数序列由 T/C 控制寄存器 TCCR2 中的标志位 WGM21 和 WGM20 确定。计数器计数行为与输出比较 OC2 的波形有紧密的关系。T/C 溢出中断标志 TOV2 根据 WGM21～WGM20 设定的工作模式来设置。TOV2 可以用于产生 CPU 中断。

2）T/C2 的工作模式 T/C2 也有 4 种不同的工作模式，即普通模式、CTC 模式、快速 PWM 模式和相位修正 PWM 模式。这 4 种模式的选择及 OC2 不同方式的输出是由

T/C2 的控制寄存器 TCCR2 的标志位 WGM21、WGM20 和 COM21、COM20 的组合决定的。由于这 4 种工作模式与 T/C0 的 4 种工作模式基本相同，所以其相关知识请参照 T/C0 的工作模式。

3）T/C2 的相关寄存器

【T/C2 控制寄存器 TCCR2】其各位定义如下:

Bit	7	6	5	4	3	2	1	0
	FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20
读/写	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ FOC2（bit7）: 强制输出比较位。FOC2 位只在 T/C2 被设置为非 PWM 模式下工作时才有效，但为了与未来器件的兼容性，在使用 PWM 时，写 TCCR2 时要对其清零。对其写“1”后，波形比较器将立即进行比较操作，比较匹配输出引脚 OC2 将按照 COM21 和 COM20 设置输出相应的电平。

> 🕮 注意:
>
> FOC2 类似于一个选通脉冲，而 OC2 的输出还取决于 COM21 和 COM20 位的设置。FOC2 不会产生任何的中断申请，也不影响计数器 TCNT2 的寄存器 OCR2 的值。

☺ WGM21、WGM20（bit3、bit6）: 波形发生模式位。这两个标志位控制 T/C2 的计数和工作方式、计数器的计数上限值及确定波形发生器的工作模式。T/C2 支持普通模式、CTC 模式及两种 PWM 模式，见表 7-7。

表 7-7　T/C2 的波形产生模式

模式	WGM21	WGM20	T/C2 的工作模式	最大值	OCR2 的更新时间	TOV2 的置位时刻
0	0	0	普通模式	0xFF	立即更新	0xFF(TOP)
1	0	1	相位修正的 PWM 模式	0xFF	0xFF(TOP)	0x00(BOTTOM)
2	1	0	CTC 模式	OCR2	立即更新	0xFF(TOP)
3	1	1	快速 PWM 模式	0xFF	0xFF(TOP)	0xFF(TOP)

☺ COM21、COM20（bit5、bit4）: 波形模式产生位。这两位用于控制比较输出引脚 OC2（PD7）的输出方式。若 COM21、COM20 中的任何一位或两位置“1”，则 OC0 的输出将覆盖 PD7 引脚的通用 I/O 端口功能，而以比较匹配输出的方式进行工作，但此时其方向控制位（DDRD.7）需要设置为 1（输出方式）。当引脚 PD7 作为 OC2 输出引脚时，其输出方式取决于 COM21、COM20 和 WGM21、WGM20 的设定，见表 7-8。

表 7-8　COM21、COM20 的位功能定义

工 作 模 式	COM21	COM20	说　　明	备　　注
0 或 2（普通模式或 CTC 模式，即非 PWM 模式）	0	0	正常的端口操作，不与 OC2 引脚物理连接	WGM21、WGM20=00 或 10
	0	1	比较匹配发生时，触发 OC2（OC2 为原 OC2 的取反）	
	1	0	比较匹配发生时，OC2 清零	
	1	1	比较匹配发生时，OC2 置位	

续表

工 作 模 式	COM21	COM20	说　　明	备　　注
3（快速 PWM 模式）	0	0	PD7 为通用 I/O 引脚（OC2 与引脚不连接）	WGM21、WGM20=11
	0	1	保留	
	1	0	比较匹配时，清零 OC2；计数值为 0xFF 时，置位 OC2	
	1	1	比较匹配时，置位 OC2；计数值为 0xFF 时，清零 OC2	
1（相位修正的 PWM 模式）	0	0	PD7 为通用 I/O 引脚（OC2 与引脚不连接）	WGM21、WGM20=01
	0	1	保留	
	1	0	向上计数过程中比较匹配时，清零 OC2 向下计数过程中比较匹配时，置位 OC0	
	1	1	向上计数过程中比较匹配时，置位 OC2 向下计数过程中比较匹配时，清零 OC2	

☺CS22、CS21、CS20（bit2~bit0）：时钟选择位。用于设定 T/C2 的时钟源，见表 7-9。

表 7-9　T/C2 时钟源的设定

CS22	CS21	CS20	说　　明
0	0	0	无时钟，T/C2 停止工作
0	0	1	$clk_{I/O}$/1（没有预分频）
0	1		$clk_{I/O}$/8（来自预分频）
0	1	1	$clk_{I/O}$/64（来自预分频）
1	0		$clk_{I/O}$/256（来自预分频）
1	0	1	$clk_{I/O}$/1024（来自预分频）
1	1	0	时钟由 T2 引脚引入，下降沿触发
1	1	1	时钟由 T2 引脚引入，上升沿触发

【T/C2 计数寄存器 TCNT2】通过 T/C2 寄存器，MCU 可以直接对计数器的 8 位数据进行读/写访问。写 TCNT2 寄存器将在下一个定时器时钟周期中阻塞比较匹配。因此，在计数器运行期间修改 TCNT2 的内容，将有可能丢失一次 TCNT2 与 OCR2 的匹配比较操作。

Bit	7	6	5	4	3	2	1	0
	TCNT2[7..0]							
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

【输出比较寄存器 OCR2】输出比较器包含一个 8 位的数据，用于与寄存器 TCNT2 中的计数值进行匹配比较。在 T/C2 运行期间，比较匹配单元一直将寄存器 TCNT2 的计数值与寄存器 OCR2 的内容进行比较。一旦 TCNT2 的计数值与 OCR2 的数据匹配相等，将产生一个输出比较匹配相等的中断申请，或者在 OC2 引脚上产生波形。其各位定义如下：

Bit	7	6	5	4	3	2	1	0
	OCR2[7..0]							
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

【定时器/计数器中断屏蔽寄存器 TIMSK】其各位定义如下:

Bit	7	6	5	4	3	2	1	0
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ OCIE2（bit7）: T/C2 输出比较匹配中断使能位。当 OCIE2 和状态寄存器 SREG 的全局中断使能位 I 置位时，T/C2 的输出比较匹配中断使能。当 T/C2 的比较匹配事件发生，即 TIFR 中的 OCF2 置位时，产生输出比较匹配中断。

☺ TOIE2（bit6）: T/C2 溢出中断使能位。当 TOIE2 和状态寄存器 SREG 的全局中断使能位 I 都置位时，T/C2 的溢出中断使能。当 T/C2 发生计数溢出，即 TIFR 中的 TOV2 置位时，产生溢出中断。

【中断标志寄存器 TIFR】中断标志寄存器 TIFR 在 I/O 空间的地址为 0x38（0x0058）。它有 8 个标志位，与 T/C2 有关的只有 OCF2 和 TOV2。其各位定义如下:

Bit	7	6	5	4	3	2	1	0
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ OCF2（bit7）: T/C2 输出比较匹配中断标志位。当 T/C2 与 OCR2（输出比较寄存器）的值匹配时，OCF2 置位（无论中断是否使能）。此位在执行中断服务程序时由硬件清零，也可以由软件写“1”清零。当 SREG 寄存器中的 I 位和 TIMSK 中的 OCIE2 及 OCF2 都置位时，中断服务程序得到执行。

☺ TOV2（bit6）: T/C2 溢出中断标志位。当 T/C2 计数溢出时，TOV2 置位（无论中断是否使能）。此位在执行中断服务程序时由硬件清零，也可以由软件写“1”清零。当 SREG 寄存器中的 I 位和 TIMSK 中的 TOIE2 都置位时，中断服务程序得到执行。

> 🕮 注意:
> 在相位修正的 PWM 模式中，当 T/C2 在 0x00 改变计数方向时，TOV2 置位。

【特殊功能 I/O 寄存器 SFIOR】其各位定义如下:

Bit	7	6	5	4	3	2	1	0
	ADTS2	ADTS1	ADTS0	—	ACME	PUD	PSR2	PSR10
读/写	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ PSR2（bit1）: T/C2 预分频器复位位。PSR2=1 时，T/C2 预分频器复位，操作完成后这一位由硬件清零；PSR2=0 时，不会引起任何操作。若内部 CPU 时钟作为 T/C2，则读该位总是返回 0。当 T/C2 工作在异步模式时，直到预分频器复位该位保持为 1。

【异步状态寄存器 ASSR】其各位定义如下:

Bit	7	6	5	4	3	2	1	0
	—	—	—	—	AS2	TCN2UB	OCR2UB	TCR2UB
读/写	R	R	R	R	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ AS2（bit3）: T/C2 异步时钟选择位。AS2=0 时，T/C2 由 I/O 时钟 clkI/O 驱动;

AS2=1 时，T/C2 由连接到 TOSC1 引脚的晶体振荡器驱动。若改变 AS2，有可能破坏 TCNT2、OCR2 与 TCCR2 的内容。

☺ TCN2UB（bit2）：T/C2 更新标志位。T/C2 工作于异步模式时，写 TCNT2 将引起 TCN2UB 置位。当 TCNT2 从暂存寄存器更新完毕后，TCN2UB 由硬件清零。TCN2UB=0，表明 TCNT2 可以写入新的数据。

☺ OCR2UB（bit1）：输出比较寄存器 2 更新标志位。T/C2 工作于异步模式时，写 OCR2 将引起 OCR2UB 置位。当 OCR2 从暂存寄存器更新完毕后，OCR2UB 由硬件清零。OCR2UB=0，表明 OCR2 可以写入新的数据。

☺ TCR2UB（bit0）：T/C2 控制寄存器更新标志位。T/C2 工作于异步模式时，写 TCCR2 将引起 OCR2UB 置位。当 TCCR2 从暂存寄存器更新完毕后，TCR2UB 由硬件清零。TCR2UB=0，表明 TCCR2 可以写入新的数据。

📖 注意：

在更新标志位置“1”时，写上述任何一个寄存器都将造成数据的破坏，并引发不必要的中断。

任务 2　用 T/C0 实现简单计数

使用单片机 PC、PB2、PB3 端口作为输出端口，外接一个 2 位 LED 数码管，PB0 外接按键 K。编写程序，每按两次 K 时，LED 数码管进行加 1 显示，计数范围为 0～99。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 7-10 所列的元器件，并完成图 7-3 所示的硬件电路图设计。

表 7-10　用 T/C0 实现简单计数所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	晶振 CRYSTAL 8MHz	电解电容 CAP-ELEC
电阻 RES	按钮 BUTTON	数码管 7SEG-MPX2-CA	三极管 NPN

软件设计

在项目六的任务 6 中通过 INT0 和 INT2 来进行加/减计数，而本任务是通过 T/C0 对外部输入信号进行计数。T/C0 可以使用中断方式对外部输入信号进行计数，每来两个外部输入信号，应产生一个溢出信号或中断信号，因此 TCNT 初值为 0xFE。其程序流程图如图 7-4 所示。流程图中的 LED 数码管显示函数请参见项目六的任务 5 中的图 6-13。

源程序

```
/*************************************************
  文件名:          用 TC0 实现简单计数.c
```

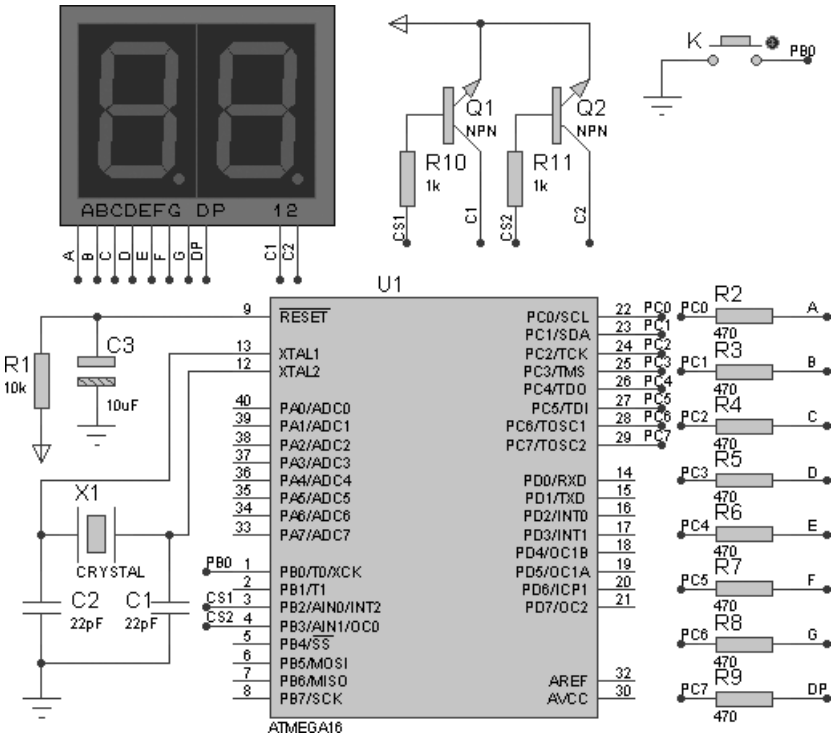


图 7-3　用 T/C0 实现简单计数的电路图

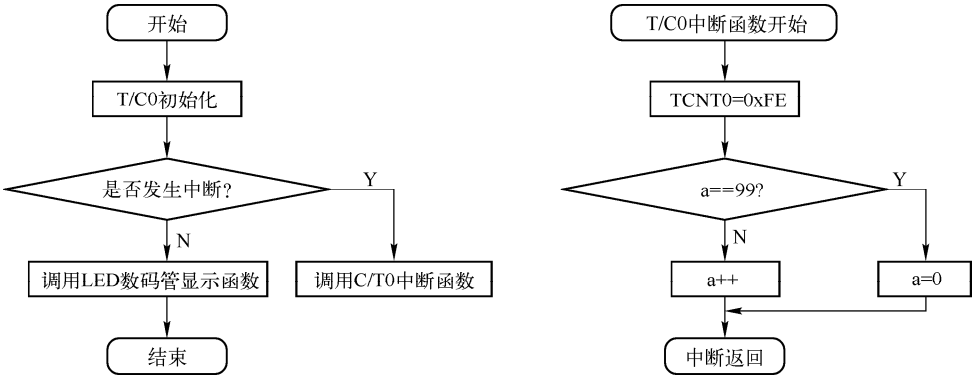


图 7-4　用 T/C0 实现简单计数的程序流程图

```
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*************************************************/
#include "mega16.h"
#define   uchar unsigned char
#define   uint unsigned int
#define   button1          PINB.0
#define   CS1               PORTB.2                          //CS1 片选
#define   CS2               PORTB.3                          //CS2 片选
#define   LED               PORTC
flash uchar tab[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,     //共阳极 LED0～F 的段码
```

```
                    0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E,0xBF,0x10};
uchar a;                                                    //计数值
uchar data_H,data_L;                                        //计数值高、低位
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void data_in(void)                                          //获取高、低位
{
  data_L=a%10;
  data_H=a/10;
}
void display(void)
{
  LED=tab[data_H];
  CS1=1;
  delay(5);
  CS1=0;
  LED=tab[data_L];
  CS2=1;
  delay(5);
  CS2=0;
}
void port_init(void)                                        //端口初始化
{
    DDRB=0xFE;                                              //PB0 端口配置为输入
   PORTB=0xFF;                                              //PB 端口初始值设置
    DDRC=0xFF;                                              //PC 端口配置为输出
   PORTC=0xFF;                                              //PC 端口初始值设置
}
void TC0_init(void)                                         //T/C0 初始化设置
{
   TCCR0=0x07;      // WGM01=0、WGM00=0，选择普通模式；COM01=0、COM00=1；PB3
                    //为通用 I/O（OC0 不与引脚相连）；CS02=1、CS01=1、CS00=1，选择外
                    //部 T0 引脚，上升沿驱动
   TCNT0=0xFE;
   OCR0=0x00;
   TIMSK=0x01;                                              //允许 T/C0 溢出中断
   #asm("sei")                                              //启动全局中断
}
interrupt [TIM0_OVF] void timer0_isr(void)                  //T/C0 计数溢出中断函数
 {
  TCNT0=0xFE;                                               //重置 TCNT0 初值
```

```
    if(a==99)
    {
      a=0;
    }
    a++;                                                    //当前计数值加 1
  }
void main(void)
{
   port_init();
   TC0_init();
   while(1)
      {
        data_in();
        display();
      }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“用 TC0 实现简单计数.COFF”文件，然后在 Proteus ISIS 中打开已创建的用 TC0 实现简单计数电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在按键 K 没有被按下的初始状态时，LED 数码管显示 0。每按 2 次按键 K 时，LED 数码管进行加 1 计数，当计数值达到 99 时，再次按下该按键则显示为 0。简单计数器的运行仿真效果如图 7-5 所示。

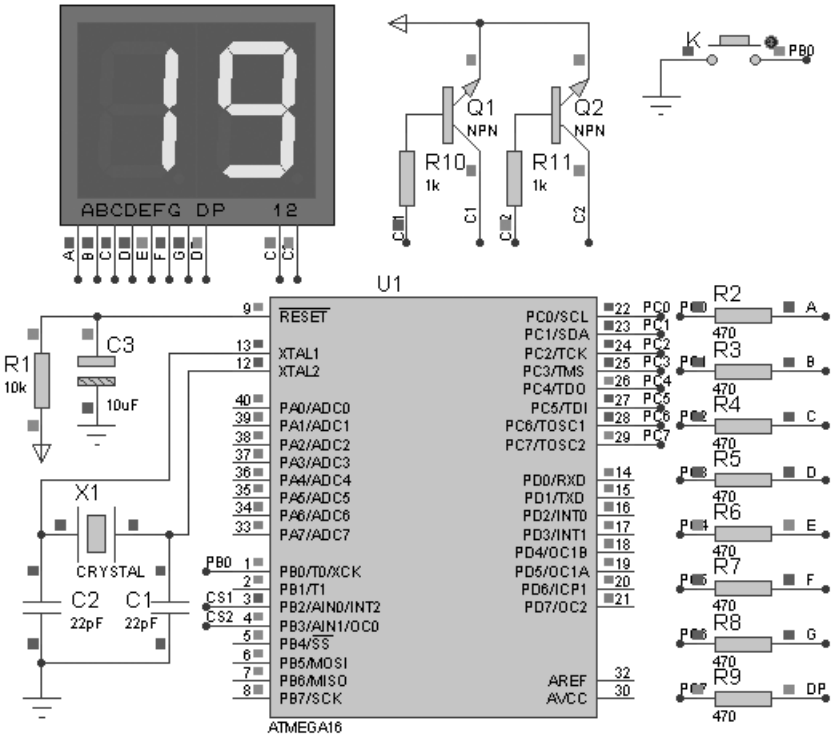


图 7-5　用 T/C0 实现简单计数运行仿真效果图

任务 3　用 T/C0 实现流水灯控制

使用单片机 PC 端口作为输出端口，外接 8 只 LED，PD2 和 PD3 分别外接按键 K1 和 K2。要求按 1 次按键 K1（INT0）启动 LED 每隔 1s 进行流水控制，按 1 次按键 K2（INT1），则 LED 暂停流水控制。



硬件设计

本设计所使用的硬件电路与项目六中的采用两个外中断实现 LED 键控移位控制完全相同，因此可参照图 6-9 绘制用 T/C0 实现流水灯控制的电路图。



软件设计

无论定时器/计数器是作为计数器使用，还是作为定时器使用，其基本的工作原理都是对一个脉冲系列信号进行计数。通常所谓的定时器，更多的情况是指其计数脉冲信号来自芯片本身的内部。由于内部的计数脉冲信号的频率是已知或固定的，所以用户可以根据需要设定计数器脉冲计数的个数，以获得一个等间隔的定时中断。

ATmega16 的定时器/计数器在内部系统时钟与计数单元之间增加了一个可设置的预分频器。利用这个预分频器，定时器/计数器可以从内部系统中获得不同频率的计数脉冲信号。表 7-11 给出了系统时钟 f_{osc} 为 8MHz 时，ATmega16 芯片本身能够提供给 T/C0 的计数脉冲信号的最高计时精度和时宽范围。

表 7-11　T/C0 的最高计时精度和时宽范围（系统时钟为 8MHz）

分频系数	计时频率（$f=f_{osc}$/分频系数）	最高计时精度（1/f）（TCNT0=0xFF）	最高时宽 t（最高计时精度×256）（TCNT0=0x00）
1	8MHz	0.125μs	32μs
8	1MHz	1μs	256μs
64	125kHz	8μs	2.048ms
256	31.25kHz	32μs	8.192ms
1024	7812.5Hz	128μs	32.768ms

PC 端口作为输出端口，因此 PC 端口的 DDRC 设置为“0xFF”。由于采用定时器/计数器 0 作为 1s 硬件延时，所以可使用 T/C0 的普通模式。假设采用分频系数为 1024，当系统时钟为 8MHz 时，需计数 30 次（1s/32.768ms≈30）。若需要较精确的定时，可采用公式 $t=(255-TCNT0)\times$分频系数$/f_{osc}$ 来计算。

在主程序中先进行 T/C0 的初始化，并启动中断，T/C0 的延时通过定时中断函数来实现。在初始化过程中使用普通模式，当分频系数为 1024 时，TCCR0=0b00000101=0x05；TCNT0=0x00；OCR0=0x00；TIMSK=0x01。除了 T/C0 初始化及中断设置外，还需对 INT0 和 INT1 进行相应设置。在 INT0 中断函数中 TIMSK=0x01，以启动 T/C0 延时；在 INT1 中断函数中 TIMSK=0x00，暂停 T/C0 延时。其程序流程图如图 7-6 所示。

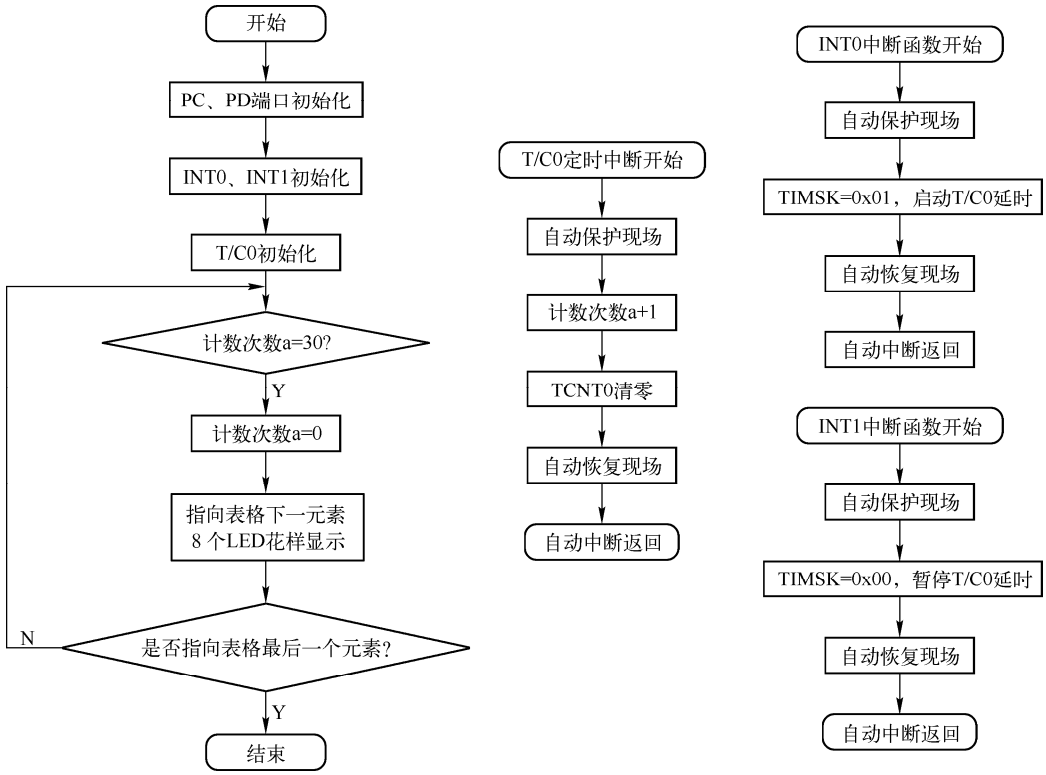


图 7-6　用 T/C0 实现流水灯控制的程序流程图

PROGRAM 源程序

```
/*****************************************************
 文件名:              用 TC0 实现流水灯控制.c
 单片机型号:          ATmega16
 晶振频率:            8.0MHz
 说明:                C/T0 作为硬件延时,INT0\INT1 启动/暂停延时
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define   button1        PIND.2
#define   button2        PIND.3
#define   LED              PORTC
flash uchar discode[]={0xFE,0xFD,0xFB,0xF7,0xEF,0xDF,0xBF,0x7F,      //左移流水
                       0xBF,0xDF,0xEF,0xF7,0xFB,0xFD,0xFE,0xFF};    //右移流水
uchar a=0;
uchar i;
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
```

```
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
interrupt[EXT_INT0] void int0_isr(void)          //INT0 中断函数
{
  delay(10);
  if(button1==0)                                 //再次确认按键，软件防抖
    {
      TIMSK=0x01;
    }
}
interrupt[EXT_INT1] void int1_isr(void)          //INT1 中断函数
{
  delay(10);
  if(button2==0)                                 //再次确认按键，软件防抖
    {
      TIMSK=0x00;
    }
}
void port_init(void)                             //端口初始化
{
     DDRC=0xFF;
    PORTC=0xFF;
     DDRD=0xF3;
    PORTD=0xFF;
}
void INT_init(void)                              //中断初始化设置
{
    MCUCR=0x0A;                                  //定义 INT0 和 INT1 为下降沿时产生中断
     GICR=0xC0;                                  //允许 INT0 和 INT1 产生中断
     #asm("sei")                                 //开启总中断
}
void TC0_init(void)                              //T/C0 初始化设置
{
  TCCR0=0x05;       //WGM01=0、WGM00=0，选择普通模式；COM01=0、COM00=0，使
                    //PB3 为通用 I/O（OC0 不与引脚相连）；CS02=1、CS01=0、CS01=1，选
                    //择 1024 分频
  TCNT0=0x00;
   OCR0=0x00;
}
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
  a++;
  TCNT0=0x00;
}
void main(void)
```

```
{
    port_init();
  TC0_init();
  INT_init();
  LED=0x00;
  while(1)
    {
      for(i=0;i<16;)
       {
         if(a==30)
           {
             LED=discode[i];
             a=0;
             i++;
           }
         else
           {;}
       }
    }
}
```



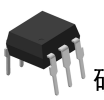
调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“用 TC0 实现流水灯控制.COFF”文件，然后在 Proteus ISIS 中打开已创建的用 TC0 实现流水灯控制电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在按键 K1 没有被按下的初始状态时，LED 全亮。按下按键 K1 后，LED 每隔 1s 进行流水显示。按下按键 K2 后，LED 暂停流水显示。



任务 4　用 T/C0 的 CTC 模式输出方波

使用 T/C0 的 CTC 模式输出一个频率约为 50Hz、占空比为 50%的方波信号。



硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 7-12 所列的元器件，并完成图 7-7 所示的硬件电路图设计。

表 7-12　用 T/C0 的 CTC 模式输出方波所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	晶振 CRYSTAL 8MHz	电阻 RES	电解电容 CAP-ELEC



软件设计

利用定时器/计数器 0 的 CTC 模式功能可以产生一个比较匹配输出，并通过控制与比较

匹配输出端口 PB3（OC0）输出方波。

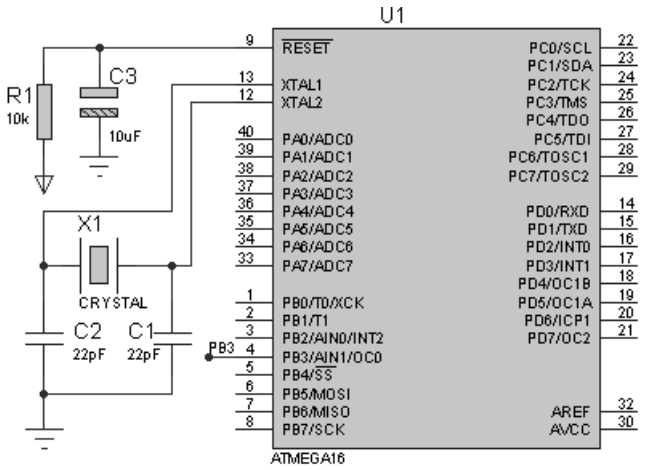


图 7-7　用 T/C0 的 CTC 模式输出方波电路图

当 T/C0 工作在 CTC 模式下时，计数器为单向加 1 计数器，一旦寄存器 TCNT0 的值与输出比较寄存器 OCR0 中的设定值相等（此时寄存器 OCR0 的值为计数上限值），就将计数器 TCNT0 清零为 0x00，同时在比较匹配输出端口 OC0 上输出相应电平，然后 TCNT0 继续向上加 1 计数。该程序流程图如图 7-8 所示。通过设置 OCR0 的值，可以方便地控制比较匹配输出的频率，也方便了外部事件计数的应用。

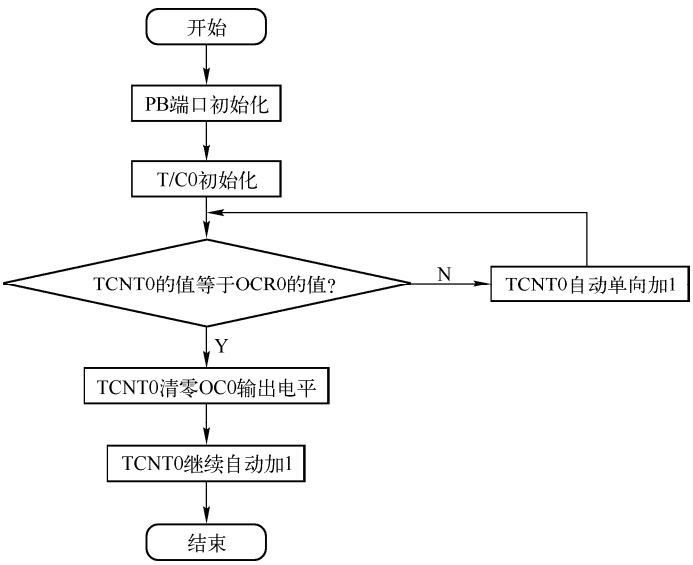


图 7-8　用 T/C0 的 CTC 模式输出方波程序流程图

在 CTC 模式下利用匹配输出单元产生波形时，其频率可由公式 $f_{OC0}=\dfrac{f_{clk_{I/O}}}{2N(1+OCR0)}$ 来计算。由于 f_{OC0}=50Hz，$f_{clk_{I/O}}$=8MHz，假如 N 取 1024，则可得 OCR0=77.125≈77。

源程序

```
/*************************************************
 文件名:          用 TC0 的 CTC 模式输出方波.c
 单片机型号:      ATmega16
 晶振频率:        8.0MHz
*************************************************/
#include "mega16.h"
#define   uchar unsigned char
#define   uint unsigned int
void port_init(void)                          //端口初始化
{
     DDRB=0xFF;                               //PB 端口配置为输出
    PORTB=0xFF;                               //PB 端口初始值设置
}
void TC0_init(void)                           //T/C0 初始化设置
{
    TCCR0=0x1D; //WGM01=1 WGM00=0 选择 CTC 模式；COM01=0 COM00=1 比较匹配时，
                //触发 OC0CS02=1 CS01=0 CS01=1 选择 1024 分频
    TCNT0=0x00;
     OCR0=77;
    TIMSK=0x02;                               //允许 T/C0 溢出中断
    #asm("sei")
}
void main(void)
{
  port_init();
  TC0_init();
  while(1)
     {
       ;
     }
}
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“用 T/C0 的 CTC 模式输出方波.COFF”文件，然后在 Proteus ISIS 中打开已创建的用 T/C0 的 CTC 模式输出方波电路图并进行相应设置。

为了在 Proteus 软件中进行仿真时能够观看到输出波形，应在 Proteus 中添加虚拟示波器。首先在 Proteus ISIS 编辑界面的工具箱中单击图标，并选择“OSCILLOSCOPE”，然后在原理图的合适位置单击鼠标左键，并将示波器通道 A 的探针与 PB3（OC0）连接起来。

单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。此时，可以通过虚拟示波器观看 PB3 输出的方波，如图 7-9 所示。从图中可看出，该方波的周期为 20ms（即频率为 50Hz），高电平与低电平的脉宽相同（即占空比为 50%），脉冲幅度为 5V。

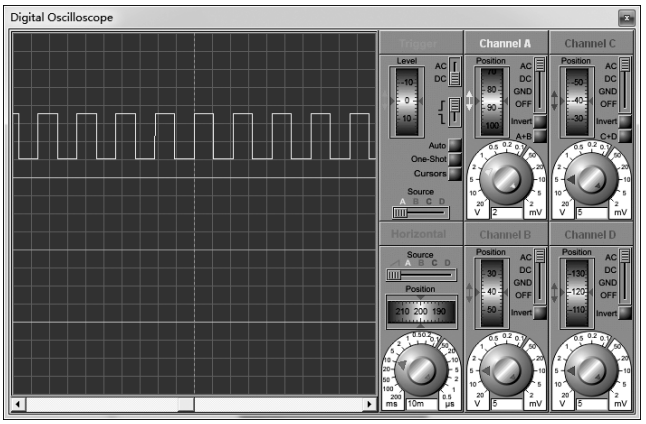


图 7-9　用 T/C0 的 CTC 模式输出的方波

任务 5　用 T/C0 的快速 PWM 模式实现调光控制

使用定时器/计数器 0 的快速 PWM 模式进行 LED 的调光控制，要求通过按键 K1 与 K2 的控制使与 PB3（OC0）连接的 LED 以渐明、渐暗来显示输出效果。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 7-13 所列的元器件，并完成图 7-10 所示的硬件电路图设计。

表 7-13　用 T/C0 的快速 PWM 模式实现调光控制所用元器件

单片机 ATmega16	晶振 CRYSTAL 8MHz	瓷片电容 CAP 22pF	电解电容 CAP-ELEC
电阻 RES	发光二极管 LED-GREEN		

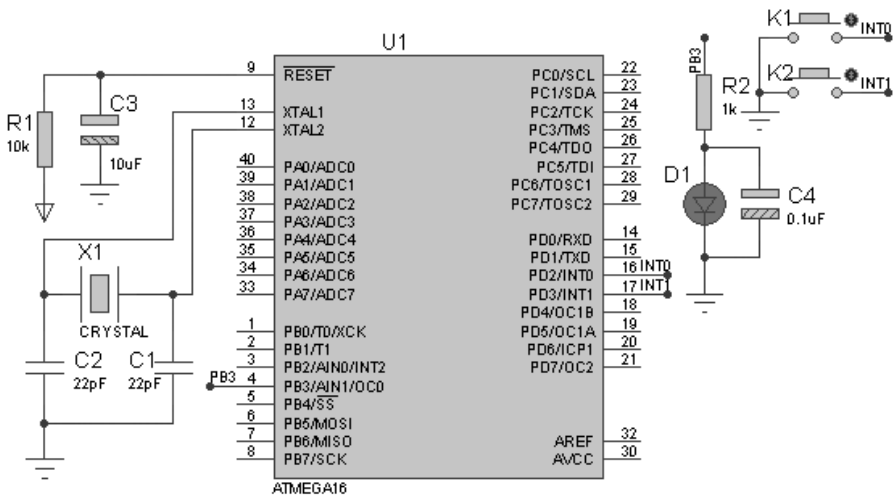


图 7-10　用 T/C0 的快速 PWM 模式实现调光控制电路图

软件设计

一个 PWM 波形的参数有频率、占空比和相位，其中频率和占空比为主要的参数。在实际应用中，除了要考虑如何正确控制和调整 PWM 波形的占空比，获得达到要求的平均电压输出外，还需要综合考虑 PWM 的周期、PWM 波形占空比的调节精度（通常用 bit 表示）、积分器的设计等。而且这些因素相互之间也是互相牵连的。

注意:

根据 PWM 的特点，在使用 AVR 定时计数器设计输出 PWM 时，应注意以下 3 点。

☺ 根据实际情况，确定 PWM 波形的频率范围。这个频率与控制的对象有关，如输出的 PWM 波用于控制灯的亮度。由于人眼不能分辨 42Hz 以上的频率，所以 PWM 的频率应高于 42Hz，否则人眼会察觉到灯的闪烁。PWM 波形的频率越高，经过积分器输出的电压也越平滑。

☺ 使用合适的占空比调节精度。PWM 波形的占空比调节精度越高，经过积分器输出的电压也越平滑。但是在相同的系统时钟频率下，如果提高占空比的调节精度，将会导致 PWM 波形的频率降低。

☺ 由于 PWM 波本身还是数字脉冲波，其中含有大量丰富的高频成分，所以在实际使用中，还需要一个好的积分器电路，如采用有源低通滤波器或多阶滤波器等，将高频成分有效地去除掉，从而获得比较好的模拟变化信号。

ATmega16 的 T/C0 具备产生 PWM 波形的功能，由于它的计数器的长度为 8 位，所以它是固定 8 位精度的 PWM 波形发生器，即 PWM 的调节精度为 8 位。因此，PWM 波形的周期只能取决于系统时钟和分频系数（即作为计数器计数脉冲的频率）。

快速 PWM 模式可以得到较高频率、相位固定的 PWM 输出，适合一些要求输出 PWM 频率较高，频率相位固定的应用场合。在快速 PWM 模式中，计数器仅工作在单向正向计数方式，计数器的上限值决定 PWM 的频率，而比较匹配寄存器 OCRn 的值决定了占空比的大小。快速 PWM 频率的计算公式为

$$f_{\mathrm{OC0PWM}}=\frac{f_{\mathrm{clk_{I/O}}}}{256N}$$

T/C0 的 PWM 模式是固定 8 位的 PWM，计数器 TCNTn 的上限值为固定的 0xFF（8 位 T/C），而比较匹配寄存器 OCRn 的值与计数器上限值之比即为占空比。

当 T/C0 工作在快速 PWM 模式下时，定时器/计数器在计数过程中，内部硬件电路会对计数值（TCNT0）和比较寄存器（OCR0）中的值进行比较，当两个值相匹配（相等）时，自动置位（清零）一个固定引脚的输出电平（OC0），而当计数器的值到达最大值时，则自动将该引脚的输出电平（OC0）清零（置位）。因此，在程序中改变比较寄存器中的值（通常在溢出中断服务中），定时器/计数器就能自动产生不同占空比的方波（PWM）信号了。而比较寄存器中值的改变可以通过 INT0 和 INT1 来实现。该程序流程图如图 7-11 所示。

源程序

```
/*************************************************
文件名:          用 TC0 的快速 PWM 模式实现调光控制.c
```

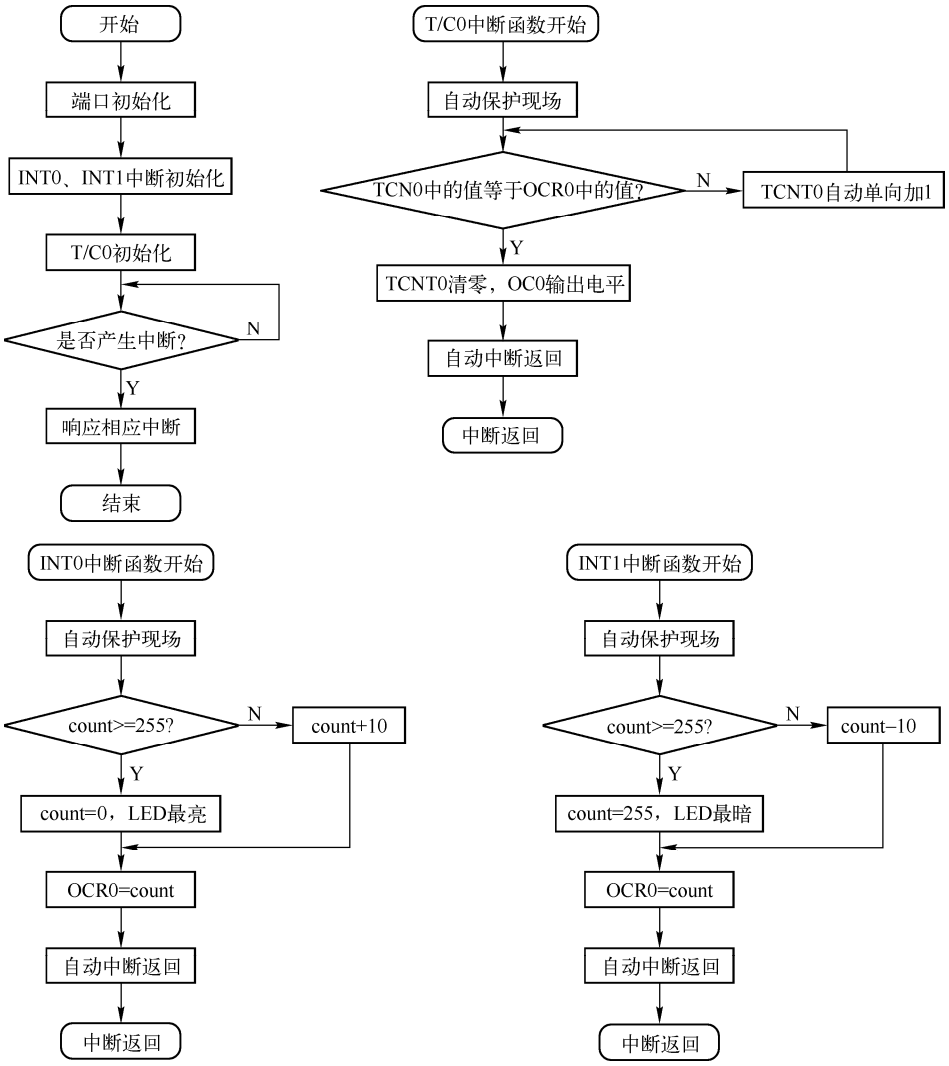


图 7-11　用 T/C0 的快速 PWM 模式实现调光控制程序流程图

```
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*************************************************/
#include "mega16.h"
#define   uchar unsigned char
#define   uint unsigned int
#define   button1          PIND.2
#define   button2          PIND.3
uchar count;
void delay(uint k)
{
   uint m,n;
   for(m=0;m<k;m++)
      {
```

```
        for(n=0;n<1140;n++)
          {;}
      }
}
void port_init(void)                          //端口初始化
{
      DDRB=0xFF;                              //PB 端口配置为输出
     PORTB=0xFF;                              //PB 端口初始值设置
      DDRD=0xF3;                              //PB 端口配置为输出
     PORTD=0xFF;
}
void TC0_init(void)                           //T/C0 初始化设置
{
    TCCR0=0x7A;                               //WGM01=1 WGM00=1 选择快速 PWM 模式；
    //COM01=1 COM00=1 比较匹配时，置位 OC0；CS02=0 CS01=1 CS01=1 选择不分频
    OCR0=0x00;
    #asm("sei")
}
interrupt[EXT_INT0] void int0_isr(void)       //INT0 中断函数
{
  delay(10);
  if(button1==0)                              //再次确认按键，软件防抖
    {
      if(count>=255)
        {
          count=0;                            //OCR=0 时，LED 最亮，然后逐渐变暗
        }
      else
        {
          count=count+10;
          OCR0=count;                         //比较匹配寄存器赋值
          delay(10);                          //延时一段时间，以观察效果
        }
    }
}
interrupt[EXT_INT1] void int1_isr(void)       //INT1 中断函数
{
  delay(10);
  if(button2==0)                              //再次确认按键，软件防抖
    {
      if(count<=0)
        {
          count=255;                          //OCR=255 时，LED 最暗，然后逐渐变亮
        }
      else
        {
          count=count-10;
          OCR0=count;                         //比较匹配寄存器赋值
```

```
            delay(10);                          //延时一段时间，以观察效果
          }
      }
}
void INT_init(void)                             //中断初始化设置
{
    MCUCR=0x0A;                                 //定义 INT0 和 INT1 为下降沿时产生中断
     GICR=0xC0;                                 //允许 INT0 和 INT1 产生中断
     #asm("sei")                                //开启总中断
}
void main(void)
{
  port_init();
  INT_init();
  TC0_init();
  while(1)
     {
       ;
     }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“用 TC0 的快速 PWM 模式实现调光控制.COFF”文件，然后在 Proteus ISIS 中打开已创建的用 TC0 的快速 PWM 模式实现调光控制电路图并进行相应设置。

为了在 Proteus 软件中进行仿真时能够观看到输出波形，应在 Proteus 中添加虚拟示波器。为了实时观察 LED 的电压值，应在 Proteus 中添加电压探针。首先在 Proteus ISIS 编辑界面的工具箱中单击图标，然后在 LED 的阳极端单击鼠标左键，即可添加电压探针。

单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。通过虚拟示波器观看 PB3 输出的 PWM 波形，如图 7-12 所示。该方波的占空比会随按键 K1 或 K2 按下的次数而实时变换。LED 也进行相应的渐明、渐暗显示，电压探针上的测量的电压值也会发生变换。

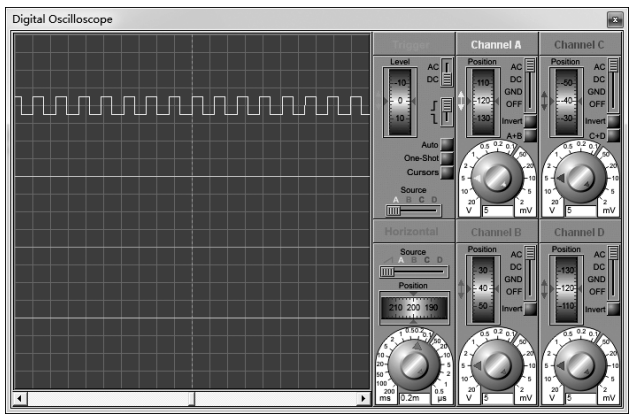


图 7-12　用 T/C0 的快速 PWM 模式实现调光控制仿真效果图

任务 6　用 T/C0 的相位修正 PWM 模式实现调光控制

使用定时器/计数器 0 的快速 PWM 模式进行 LED 的调光控制，要求通过按键 K1 与 K2 的控制使与 PB3（OC0）连接的 LED 以渐明、渐暗来显示输出效果。

硬件设计

本实例的电路与任务 5 中的图 7-10 完全一样，都是通过比较匹配输出端口 PB3（OC0）上的电平变化来使 LED 产生相应变化的。

软件设计

相位修正 PWM 模式可以产生高精度相位可调的 PWM 波形。由于相位修正 PWM 模式采用了双向计数方式，所以它产生的 PWM 波形的频率比快速 PWM 模式的频率低。

相位修正 PWM 模式的输出频率比较低，此时计数器工作在双向计数方式。同样，计数器的上限值决定了 PWM 的频率，比较匹配寄存器的值决定了占空比的大小。PWM 频率的计算公式为 $f_{\mathrm{OC0PCPWM}}=\dfrac{f_{\mathrm{clk_{I/O}}}}{510N}$。

当 T/C0 工作在此模式下时，计数器为双向计数器：从 0x00 一直加到 0xFF，在下一个计数脉冲到达时，改变计数方向，从 0xFF 开始减 1 计数到 0x00。设置反向比较匹配输出 COM01、COM00=11 时：在正向加 1 过程中，当 TCNT0 的计数值与 OCR0 的值相同匹配时，置位 OC0；在反向减 1 过程中，当计数器 TCNT0 的值与 OCR0 相同时清零 OC0。通过设置寄存器 OCR0 的值，可以获得不同占空比的脉冲波形。其程序流程图请参见图 7-11，在此不再赘述。

源程序

```
/*****************************************************
  文件名:          用 TC0 的相位修正 PWM 模式实现调光控制.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#define   uchar unsigned char
#define   uint unsigned int
#define   button1        PIND.2
#define   button2        PIND.3
uchar count;
void delay(uint k)
{
   uint m,n;
   for(m=0;m<k;m++)
      {
```

```
        for(n=0;n<1140;n++)
          {;}
      }
}
void port_init(void)                          //端口初始化
{
     DDRB=0xFF;                               //PB 端口配置为输出
    PORTB=0xFF;                               //PB 端口初始值设置
     DDRD=0xF3;                               //PB 端口配置为输出
    PORTD=0xFF;
}
void TC0_init(void)                           //T/C0 初始化设置
{
    TCCR0=0x72;      //WGM01=0，WGM00=1，选择相位修正 PWM 模式
                     //COM01=1，COM00=1，向上计数比较匹配时，置位 OC0；向下计数比较
                     //匹配时，清零 OC0
                     //CS02=0，CS01=1，CS01=0，选择 8 分频
    OCR0=0x00;
    #asm("sei")
}
interrupt[EXT_INT0] void int0_isr(void)       //INT0 中断函数
{
  delay(10);
  if(button1==0)                              //再次确认按键，软件防抖
    {
      if(count>=255)
        {
          count=0;                            //OCR=0 时，LED 最亮，然后逐渐变暗
        }
      else
        {
          count=count+10;
          OCR0=count;                         //比较匹配寄存器赋值
          delay(10);                          //延时一段时间，以观察效果
        }
    }
}
interrupt[EXT_INT1] void int1_isr(void)       //INT1 中断函数
{
  delay(10);
  if(button2==0)                              //再次确认按键，软件防抖
    {
      if(count<=0)
        {
          count=255;                          //OCR=255 时，LED 最暗，然后逐渐变亮
        }
      else
        {
          count=count-10;
          OCR0=count;                         //比较匹配寄存器赋值
          delay(10);                          //延时一段时间，以观察效果
        }
```

```
    }
}
void INT_init(void)                          //中断初始化设置
{
    MCUCR=0x0A;                              //定义 INT0 和 INT1 为下降沿时产生中断
     GICR=0xC0;                              //允许 INT0 和 INT1 产生中断
     #asm("sei")                             //开启总中断
}
void main(void)
{
  port_init();
  INT_init();
  TC0_init();
  while(1)
    {
      ;
    }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“用 TC0 的相位修正 PWM 模式实现调光控制.COFF”文件，然后在 Proteus ISIS 中打开已创建的用 TC0 的相位修正 PWM 模式实现调光控制电路图并进行相应设置。

为了在 Proteus 软件中进行仿真时能够观看到输出波形，应在 Proteus 中添加虚拟示波器。为了实时观察 LED 的电压值，应在 Proteus 中添加电压探针。首先在 Proteus ISIS 编辑界面的工具箱中单击图标，然后在 LED 的阳极端单击鼠标左键，即可添加电压探针。

单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。通过虚拟示波器观看 PB3 输出的 PWM 波形及 LED 阳极的波形，如图 7-13 所示。该方波的占空比会随按键 K1 或 K2 按下的次数而实时变换。LED 也进行相应的渐明、渐暗显示，电压探针上的测量的电压值也会发生变换。

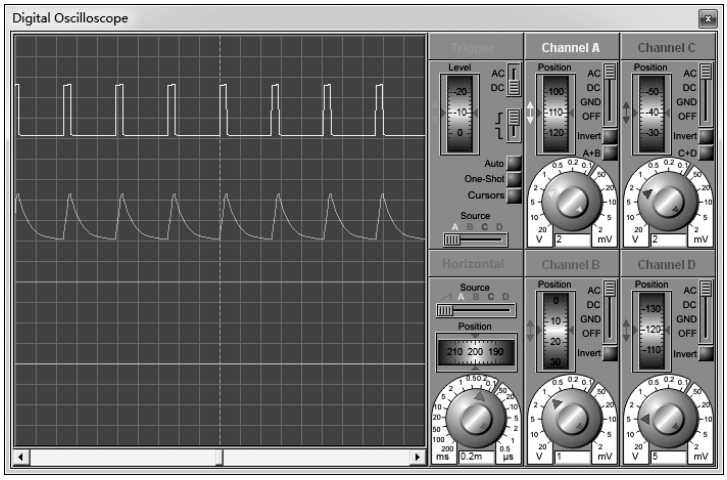


图 7-13　虚拟示波器测量 PB3 输出的 PWM 波形及 LED 阳极的波形

任务 7　用 T/C1 计数功能实现数控分频

使用定时器/计数器 1 来设计一个可选分频器。通过按下按键次数的不同，可输出 4 分频、6 分频、8 分频，直到 20 分频，并显示分频系数。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 7-14 所列的元器件，并完成图 7-14 所示的硬件电路图设计。

表 7-14　用 T/C1 计数功能实现数控分频所用元器件

单片机 ATmega16	晶振 CRYSTAL 8MHz	瓷片电容 CAP 22pF	电解电容 CAP-ELEC
电阻 RES	555 多谐振荡器	共阳极 LED 数码管　7SEG-MPX2-CA-BLUE	

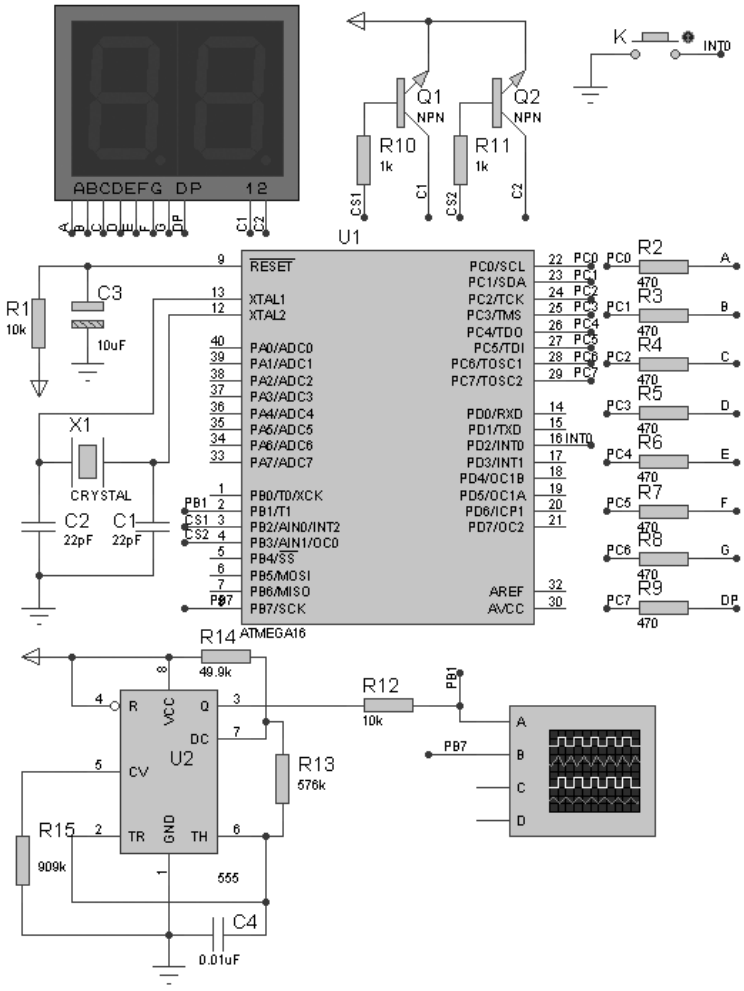


图 7-14　用 T/C1 计数功能实现数控分频电路图

软件设计

本设计实质上是设计一个 2N 分频系统。要设计一个 2N 分频系统，首先应考虑使用 T/C1 的普通模式（WGM13～WGM10=0000），采用 T1 上升沿触发（CS12～CS10=111），并设置 TCNT1（TCNT1L 和 TCNT1H）的初值。若 TCNT1=0xFFFE 时，则可以在 PD7 引脚上得到 T/C1 的 4 分频输出信号；若 TCNT1=0xFFFD 时，则可以在 PD7 引脚上得到 T/C1 的 6 分频输出信号……

要实现通过按下按键次数的不同输出不同分频信号，则可通过 INT0（或 INT1）外部中断来进行按键次数的计数。如果 T/C1 的计数也采用中断方式，则本设计需使用两个中断。其程序流程图如图 7-15 所示。

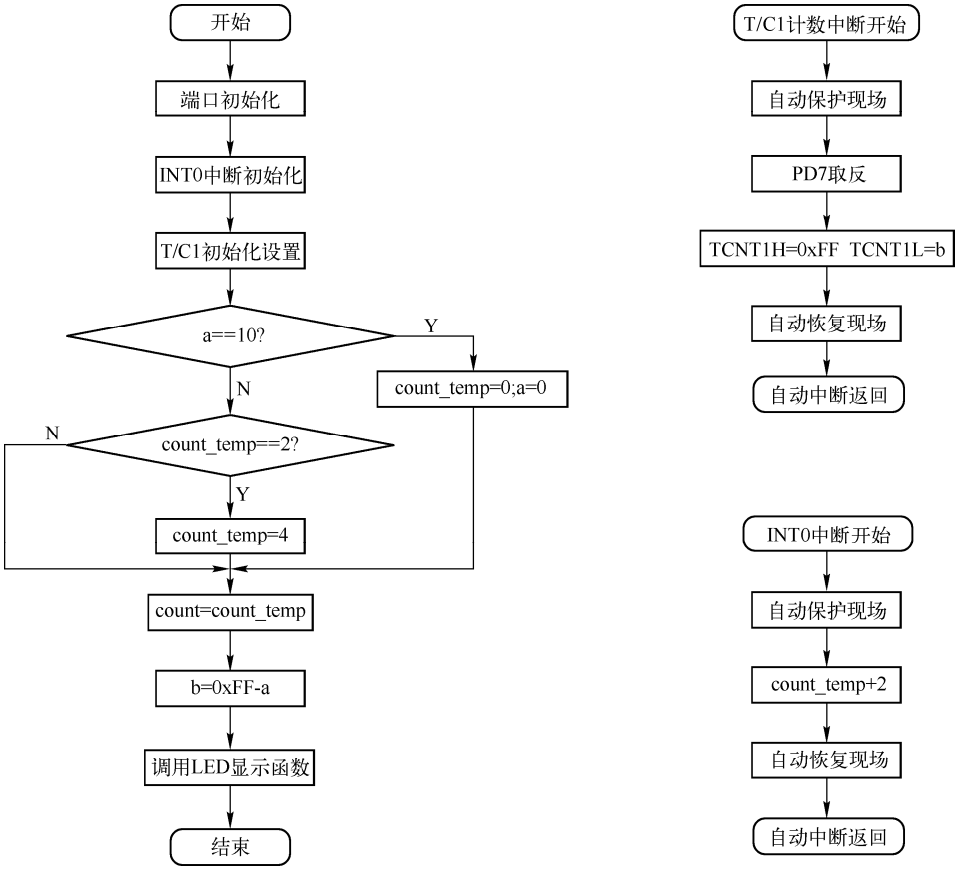


图 7-15 用 T/C1 计数功能实现数控分频程序流程图

源程序

```
/*****************************************************
  文件名:          用 TC1 计数功能实现数控分频.C
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
```

```
#define uchar unsigned char
#define uint unsigned int
#define   CS1                PORTB.2                    //CS1 片选
#define   CS2                PORTB.3                    //CS2 片选
#define   button             PORTD.2                    //与按钮连接
#define   LED                PORTC
uint a=0,b;
uchar count,count_temp;
uchar data_L,data_H;
flash uchar tab[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,    //共阳极 LED0～F 的段码
                   0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E};
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void data_in(void)                                        //获取高、低位
{
  data_L=count%10;
  data_H=count/10;
}
void display(void)
{
  LED=tab[data_H];
  CS1=1;
  delay(5);
  CS1=0;
  LED=tab[data_L];
  CS2=1;
  delay(5);
  CS2=0;
}
interrupt [EXT_INT0] void int0_isr(void)                  //INT0 中断函数
 {
    delay(10);
    if (button==0)
      {
        a++;
        count_temp=count_temp+2;
        TIMSK=0x04;
      }
  }
 interrupt [TIM1_OVF] void timer1_isr(void)               //T/C1 计数溢出中断函数
 {
```

```
    TCNT1H=0xFF;
    TCNT1L=b;
    PORTB.7=~PORTB.7;
  }
void port_init(void)                    //端口初始化
{
     DDRB=0xFD;                         //PB1 端口配置为输入，脉冲由 T1（PORTB.1）输入
    PORTB=0xFF;                         //PB 端口初始值设置
     DDRC=0xFF;                         //PC 端口配置为输出
    PORTC=0xFF;                         //PC 端口初始值设置
}
void TC1_init(void)                     //T/C1 初始化设置
{
   TCCR1A=0x00;
    TCCR1B=0x07;
    TCNT1H=0xFF;
    TCNT1L=b;
     TIMSK=0x04;
}
void INT_init(void)                     //中断初始化设置
{
    MCUCR=0x0A;
     GICR=0x40;
     #asm("sei")                        //开启总中断
}

void main(void)
  {
    port_init();
    TC1_init();
    INT_init();
    while(1)
       {
          if(a==10)                     //判断当前按键次数是否等于 10
             {                          //当前次数等于 10
           count_temp=0;                //count 清零
                a=0;
             }
           if (count_temp==2)
             {
               count_temp=4;
             }
           else
             {
              count=count_temp;
             }
          b=255-a;
```

```
            data_in();
            display();                          //调用 LED 数码管显示当前计数值
        }
    }
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“用 TC1 计数功能实现数控分频.COFF”文件，然后在 Proteus ISIS 中打开已创建的用 T/1 计数功能实现数控分频电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试状态下，两位共阳极 LED 数码管显示分频系数。当其显示到 20 后，又会从 00 开始进行下一轮的显示。PD7 输出的波形在虚拟示波器中进行显示，在虚拟示波器中的上方显示的方波是 T1 端输入的波形，即通过 555 振荡电路形成的方波，下方显示的方波是通过 6 分频后得到的方波，仿真效果如图 7-16 所示。

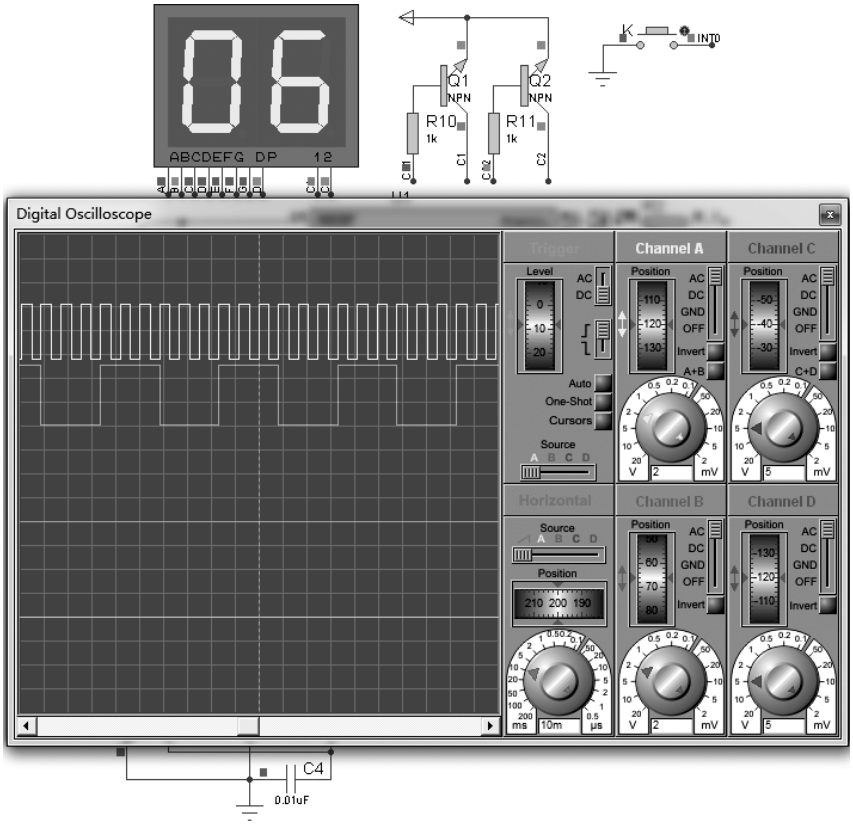


图 7-16　用 TC1 计数功能实现数控分频仿真效果图

任务 8　用 T/C1 实现预置值计时器的设计

使用单片机 PC 端口、PB4～PB7 作为输出端口外接 4 位 LED 共阳数码管，PD2 和 PD3

分别外接按键 K1 和 K2。按键 K1 作为预置计时输入端，按键 K2 作为 T/C1 计时设置与启动输入端。要求奇数次按下按键 K2 后，允许按键 K1 输入预置计时值，T/C1 不能倒计时；偶数次按下按键 K2 后，启动 T/C1 按预置计时值进行倒计时，而按键 K1 此时不能进行预置值。预置时间范围为 00～99s。



硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 7-15 所列的元器件，并完成图 7-17 所示的硬件电路图设计。

表 7-15　用 T/C1 实现预置值计时器所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	晶振 CRYSTAL 8MHz	电解电容 CAP-ELEC
电阻 RES	按钮 BUTTON	数码管 7SEG-MPX4-CA	三极管 NPN

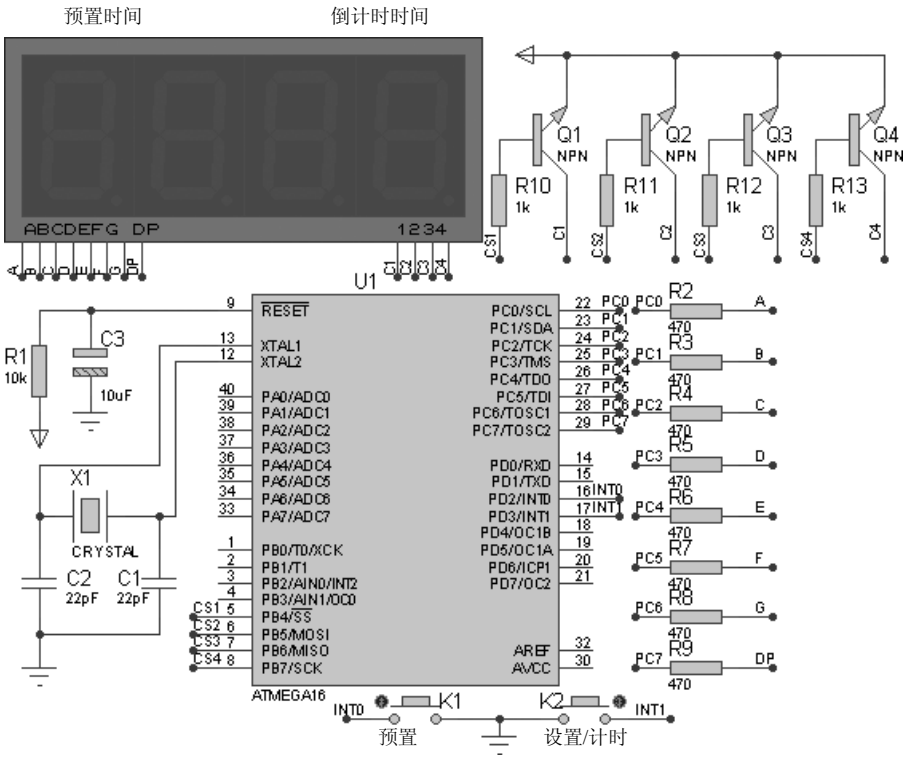


图 7-17　用 T/C1 实现预置值计时器电路图



软件设计

定时器/计数器 1 为 16 位定时器/计数器，在系统时钟为 8MHz 条件下，不需使用辅助软件计数器，就可以直接设计一个时间长达 8.388608s 的定时器，这是一般 8 位单片机做不到的。

用 T/C1 精确延时 1s 时，可采用公式 t=(65535−TCNT1) × 分频系数/f_{osc} 来计算。与系统频率 f_{osc}=8MHz，t=1s 时，若采用分频系数为 256，可得出 TCNT1=65535−8000000÷256=

65535−31250=34285=0x85ED，即 TCNT1H=0x85，TCNT1L=0xED。

在主程序中需对 INT0、INT1 和 T/C1 的初始化，为了显示时间，在主程序中只需调用 LED 显示函数即可。time1 作为 T/C1 倒计时值，time2 作为预置计时值，a 作为按键 K2 按下的奇偶次数判断位。

通过 T/C1 的定时中断函数来实现倒计时，每次定时到 1s 时，将产生一次中断，在此中断函数中需对 TCNT1 重新赋值，同时计时值 times 减 1。由于采用了 4 位一体的 LED 数码管显示数据，所以在 LED 显示函数中需将 time1 和 time2 值分别进行十位与个数值的分离。该设计的主要程序流程图如图 7-18 所示。

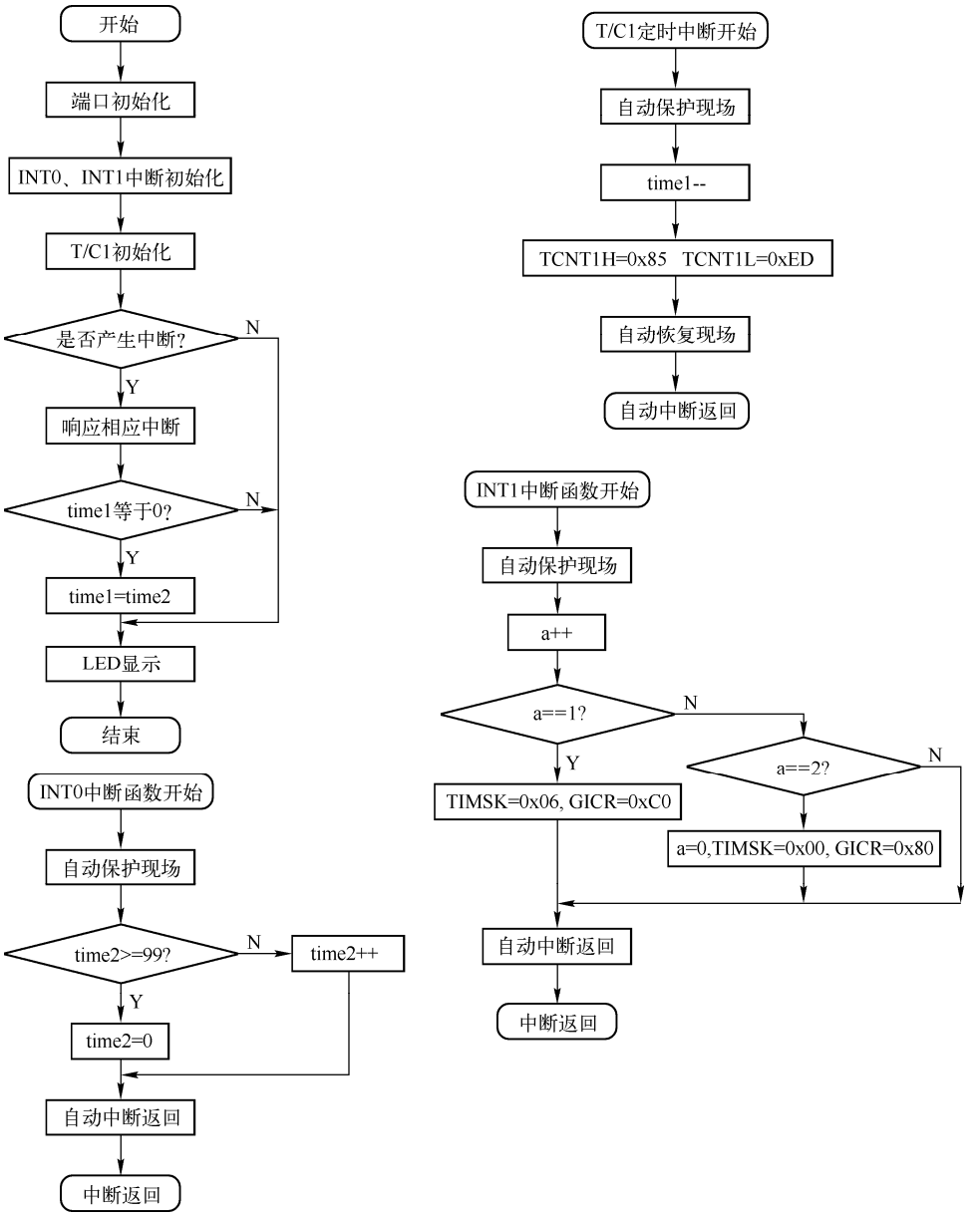


图 7-18　用 T/C1 实现预置值计时器程序流程图

注意：

在编写程序时，T/C1 和 INT0 的中断允许位由 INT1 的中断函数控制。

源程序

```
/*****************************************************
 文件名:          用 TC1 实现预置值计时器.C
 单片机型号:      ATmega16
 晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define   CS1         PORTB.4
#define   CS2         PORTB.5
#define   CS3         PORTB.6
#define   CS4         PORTB.7
#define   button1       PIND.2        //输入预置值
#define   button2       PIND.3        //暂停/启动 T/C1 计时
#define   LED         PORTC
uint a;                               //K2 按下次数
uint time1,time2;                     //time1 为倒计时值；time2 为预置计时值
uint time1_sech,time1_secl;
uint time2_sech,time2_secl;
flash uchar tab[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,  //共阳极 LED0～F 的段码
                   0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E,0xBF,0x10};
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void time_in(void)
{
  time1_sech=time1/10;
  time1_secl=time1%10;
  time2_sech=time2/10;
  time2_secl=time2%10;
}
void display(void)                    //LED 数码管显示函数
 {
    PORTB=0x0F;
    LED=tab[time2_sech];
    CS1=1;
```

```
    delay(1);
    CS1=0;
    LED=tab[time2_secl];
    CS2=1;
    delay(1);
    CS2=0;
    LED=tab[time1_sech];
    CS3=1;
    delay(1);
    CS3=0;
    LED=tab[time1_secl];
    CS4=1;
    delay(1);
    CS4=0;
 }

interrupt[EXT_INT0] void int0_isr(void)                    //INT0 中断函数
{
  delay(10);
  if(button1==0)                                           //再次确认按键，软件防抖
    {
      if(time2==99)
        {
          time2=0;
        }
      else
        {
          time2++;
        }
    }
}
interrupt[EXT_INT1] void int1_isr(void)                    //INT1 中断函数
{
  delay(10);
  if(button2==0)                                           //再次确认按键，软件防抖
    {
     a++;
      if(a==1)
        {
          TIMSK=0x00;                                      //T/C1 停止计时
          GICR=0xC0;                                       //允许 INT0 和 INT1 产生中断
        }
      if(a>=2)
        {
          a=0;
          TIMSK=0x06;                                      //T/C1 启动计时
```

```
            GICR=0x80;                                  //禁止 INT0 产生中断，允许 INT1 中断
          }
      }
}
interrupt [TIM1_OVF] void timer0_ovf_isr(void)
 {
    time1--;
    TCNT1H=0x85;
    TCNT1L=0xED;
 }
 void port_init(void)
 {
     DDRB=0xFC;
    PORTB=0xFF;
     DDRC=0xFF;
    PORTC=0xFF;
     DDRD=0xF3;
    PORTD=0xFF;
 }
void INT_init(void)                                     //中断初始化设置
{
    MCUCR=0x0A;                                         //定义 INT0 和 INT1 为下降沿时产生中断
     GICR=0x80;                                         //禁止 INT0 产生中断，允许 INT1 中断
     #asm("sei")                                        //开启总中断
}
void TC1_init(void)                                     //T/C1 初始化设置
{
  TCCR1A=0x00;   //WGM13～WGM10=0000，普通模式；COM1A1、COM1A0=00，普通端口操作
                 //COM1B1、COM1B0=00，普通端口操作
    TCCR1B=0x04; //CS12～CS10=100，256 分频
    TCNT1H=0x85;
  TCNT1L=0xED;
}
 void main(void)
 {
    port_init();
    INT_init();
    TC1_init();
    while(1)
       {
         if(time1==0)
            {time1=time2;}
         time_in();
         display();
       }
 }
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“用 TC1 实现预置值计时器.COFF”文件，然后在 Proteus ISIS 中打开已创建的用 TC1 实现预置值计时器电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在初始状态下，4 位 LED 数码管均显示为 0，奇数次按下按键 K2 后，通过多次按下按键 K1 输入预置计时值，数码管的前两位显示预置值，而后两位显示为其他数值；偶数次按下按键 K2 后，启动 T/C1 按预置计时值进行倒计时，而按键 K1 此时不能进行预置值的设置，数码管的前 2 位显示预置值，而后两位每隔 1s 进行倒计时显示。当后两位显示为 0 后，又重新从预置值开始倒计时显示，仿真效果如图 7-19 所示。

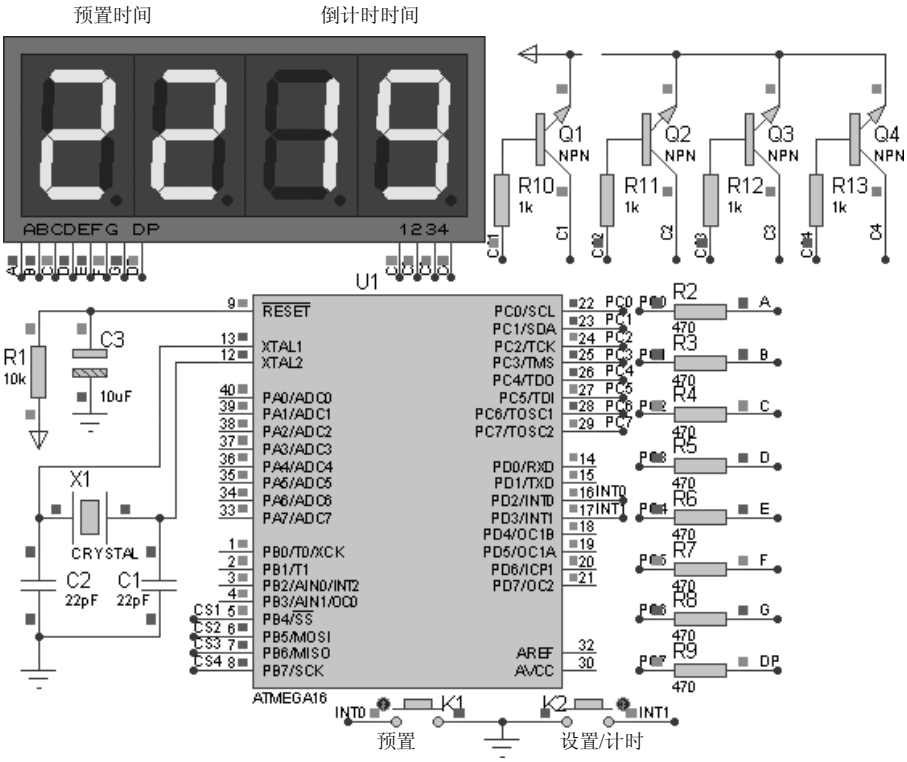


图 7-19　用 T/C1 实现预置值计时器仿真效果图

任务 9　用 T/C1 的 CTC 模式实现 LED 滚动闪烁控制

使用 T/C1 的 CTC 模式进行延时，设计一个 LED 滚动闪烁控制系统。要求当前 LED 闪烁几次后灭,下一个 LED 再闪烁，达到滚动闪烁的效果。

硬件设计

本任务可以采用项目四的任务 3 中图 4-7 的电路图，所以本设计的电路图请参见图 4-7。

软件设计

当定时器/计数器 1 工作在 CTC 模式下时，利用其比较匹配输出的性能，可直接进行延时控制。在 CTC 模式（WGM13～WGM10=0100）下，当 TCNT1 等于 OCR1A 时，计数器清零。要实现 1s 的硬件延时，定时器/计数器 1 应工作在 CTC 模式下，OCR1A 的初值可按如下公式计算：$t=f_{osc}$/分频数−OCR1A，由于 t=1s，f_{osc}=8MHz，若分频数为 256，则 OCR1A 等于 0x7A12。

LED 滚动闪烁控制程序的编写思路：T/C1 每次产生 1s 硬件延时中断时，t 加 1，i 也加 1。其中 t 作为滚动时间切换控制，i 作为闪烁时间控制。当 t 等于 40 时，t 清零，j 加 1（j 作为 LED 滚动选择控制），如果 j 等于 8，则 j 清零，否则 j 继续加 1。当 i 等于 5 时，i 清零，此时根据 j 值使相应 LED 状态取反，实现闪烁控制，而其他 LED 熄灭。例如，j 等于 4 时，如果 i 等于 5，则 D4 闪烁，而其他 LED 熄灭。其程序流程图如图 7-20 所示。

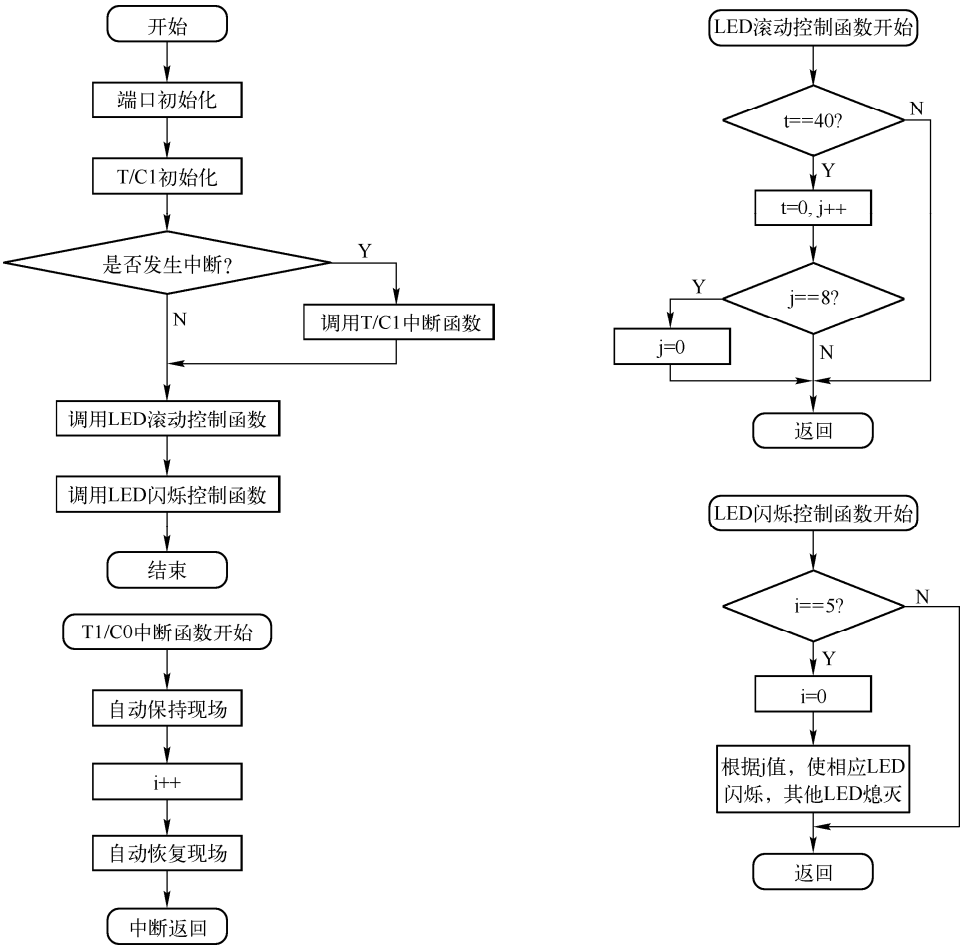


图 7-20　用 T/C1 的 CTC 模式实现 LED 滚动闪烁控制程序流程图

源程序

```
/*****************************************************
文件名:          用 TC1 的 CTC 模式实现 LED 滚动闪烁控制.C
```

```
 单片机型号:      ATmega16
 晶振频率:        8.0MHz
*************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define   D0              PORTC.0
#define   D1              PORTC.1
#define   D2              PORTC.2
#define   D3              PORTC.3
#define   D4              PORTC.4
#define   D5              PORTC.5
#define   D6              PORTC.6
#define   D7              PORTC.7
uint i,j,t;
interrupt [TIM1_COMPA] void timer1_isr(void)        // T/C1 CTC 中断比较函数
 {
   i++;
   t++;
 }
void LED_count(void)
{
   if(t==40)
      {
        t=0;
        j++;
        if(j==8)
           {
             j=0;
           }
      }
}
void LED_display(void)
{
  if(i==5)
     {
       i=0;
       switch(j)
          {
            case 0:                                  //控制 D0 闪烁，其他熄灭
              D0=~D0;D1=1;D2=1;D3=1;
              D4=1;D5=1;D6=1;D7=1;
              break;
            case 1:                                  //控制 D1 闪烁，其他熄灭
              D0=1;D1=~D1;D2=1;D3=1;
              D4=1;D5=1;D6=1;D7=1;
              break;
            case 2:                                  //控制 D2 闪烁，其他熄灭
```

```
                D0=1;D1=1;D2=~D2;D3=1;
                D4=1;D5=1;D6=1;D7=1;
                break;
            case 3:                                    //控制 D3 闪烁，其他熄灭
                D0=1;D1=1;D2=1;D3=~D3;
                D4=1;D5=1;D6=1;D7=1;
                break;
            case 4:                                    //控制 D4 闪烁，其他熄灭
                D0=1;D1=1;D2=1;D3=1;
                D4=~D4;D5=1;D6=1;D7=1;
                break;
            case 5:                                    //控制 D5 闪烁，其他熄灭
                D0=1;D1=1;D2=1;D3=1;
                D4=1;D5=~D5;D6=1;D7=1;
                break;
            case 6:                                    //控制 D6 闪烁，其他熄灭
                D0=1;D1=1;D2=1;D3=1;
                D4=1;D5=1;D6=~D6;D7=1;
                break;
            case 7:                                    //控制 D7 闪烁，其他熄灭
                D0=1;D1=1;D2=1;D3=1;
                D4=1;D5=1;D6=1;D7=~D7;
                break;
          }
      }
}
void port_init(void)
{
    DDRC=0xFF;
    PORTC=0xFF;                                        //PORTC 端口与 LED 段码相连
}
void TC1_init(void)
{
    TCCR1A=0x00;
    TCCR1B=0x0C;
    TCNT1H=0x00;
    TCNT1L=0x00;
    OCR1AH=0x7A;
    OCR1AL=0x12;
     TIMSK=0x10;
      #asm("sei")                                      //启动全局中断
}
void main(void)
 {
    port_init();
    TC1_init();
    while(1)
       {
```

```
            LED_count();
            LED_display();
        }
    }
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“用 TC1 的 CTC 模式实现 LED 滚动闪烁控制.COFF”文件，然后在 Proteus ISIS 中打开已创建的用 TC1 的 CTC 模式实现 LED 滚动闪烁控制电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试状态下，可以看到 D0 闪烁后，D1 再闪烁，而后 D2 闪烁……D7 闪烁，然后 D0 闪烁，如此循环。如果将 t 的判断值改大时，可以看到 LED 闪烁次数增大；如果将 i 的判断值改大时，LED 闪烁次数减少。

任务 10　用 T/C1 的快速 PWM 模式输出占空比可调方波

使用 T/C1 的快速 PWM 模式设计一个占空比可调的 PWM 波形。要求单片机刚启动时，占空比为 0，以后每按一次按键，占空比以 10%递增，当占空比达到 90%，再按一次按键时，LED 显示为 99%。若再次按下按键后，又将占空比置为 0，如此循环。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 7-16 所列的元器件，并完成如图 7-21 所示的硬件电路图设计。

表 7-16　用 T/C1 的快速 PWM 模式输出占空比可调方波所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	晶振 CRYSTAL 8MHz	电解电容 CAP-ELEC
电阻 RES	按钮 BUTTON	数码管 7SEG-MPX4-CA	三极管 NPN

软件设计

使用 T/C1 制作 PWM 的一种方法是，利用输出比较寄存器在每次发生匹配时改变被重新加载的递增值，从而产生 PWM 波形。

分辨率的高低决定了 PWM 控制的优劣或精度。T/C1 有 8 位、9 位和 10 位 3 种分辨率的快速 PWM 模式。在 8 位分辨率的模式下，PWM 控制精度可以达到 1/256；在 9 位分辨率的模式下，PWM 控制精度可以达到 1/512；在 10 位分辨率的模式下，PWM 控制精度可以达到 1/1024。

PWM 模式输出波形的实际占空比是由加载到定时器/计数器的输出比较寄存器的数值决定的。在正常 PWM 模式下，计数器向下计数时遇到匹配，就将输出位设置为高电平；如果是向上计数时遇到匹配，就将输出位的值清除，即设置为低电平。本设计采用 8 位快速 PWM，如果加载到输出比较寄存器的数值等于计数上限值的 10%时，就可以产生占空比为

10%的波形；如果等于上限值的 20%时，就可以产生占空比为 20%的波形……

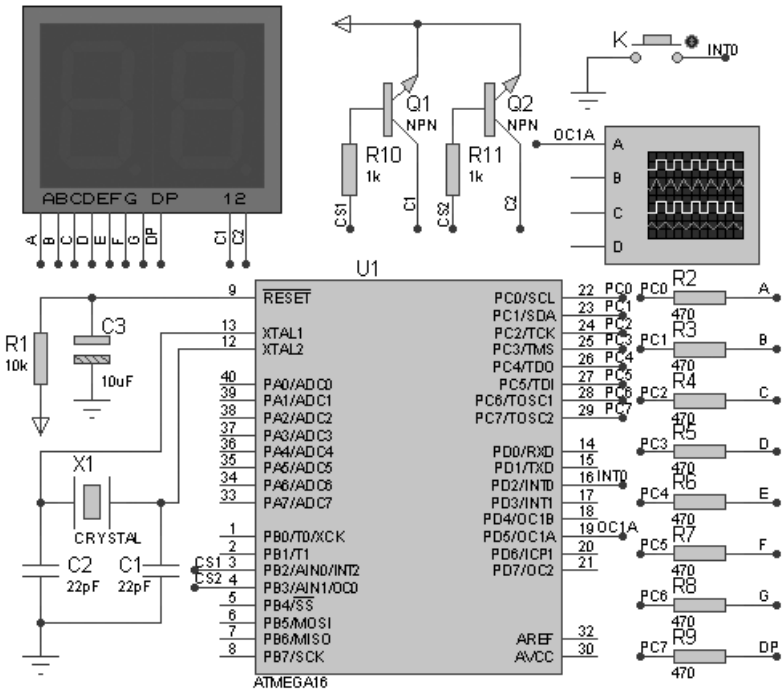


图 7-21　用 T/C1 的快速 PWM 模式输出占空比可调方波电路图

在程序中要实现占空比的选择，可以使用外部中断进行控制。每次按下按键时，计数值加 1，然后在主程序中通过判断计数值的大小给比较寄存器和 LED 数码管的显示内容赋值。其程序流程图如图 7-22 所示。

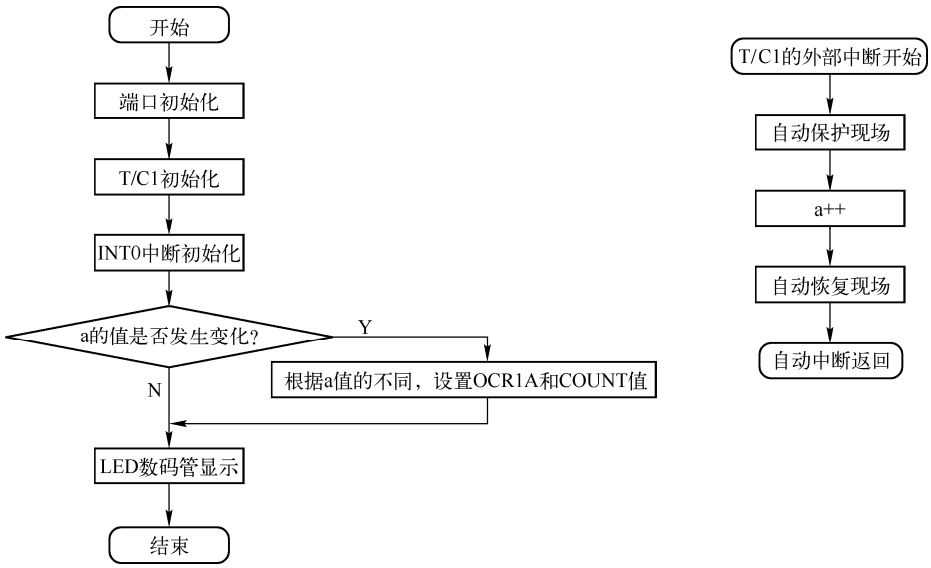


图 7-22　用 T/C1 的快速 PWM 模式输出占空比可调方波程序流程图

PROGRAM 源程序

```
/*****************************************************
  文件名:          用 TC1 的快速 PWM 模式输出占空比可调方波.C
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define   CS1          PORTB.2                  //与十位 LED 片选端连接
#define   CS2          PORTB.3                  //与个位 LED 片选端连接
#define   button       PIND.2                   //与按键连接
#define   LED          PORTC
uchar count;
uchar a,a_temp;
uchar counth,countl;
flash uchar tab[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,  //共阳极 LED0～F 的段码
                   0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E};
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void data_in(void)
{
   counth=count/10;                            //十位分离
   countl=count%10;                            //个位分离
}
void display(void)                             //LED 数码管显示函数
 {
    LED=tab[counth];                           //送十位的段码值到 PORTC
    CS1=1;                                     //显示 LED 数码管的十位
    delay(5);                                  //保持一定时间
    CS1=0;                                     //不显示 LED 数码管的十位
    LED=tab[countl];                           //送个位的段码值到 PORTC
    CS2=1;                                     //显示 LED 数码管的个位
    delay(5);                                  //保持一定时间
    CS2=0;                                     //不显示 LED 数码管的个位
 }
interrupt [EXT_INT0] void int0_isr(void)       //INT0 中断函数
 {
   a++;
```

```
}
void port_init(void)
{
    DDRB=0xFF;
   PORTB=0xFF;
    DDRC=0xFF;
   PORTC=0xFF;
    DDRD=0xFB;
   PORTD=0xFF;
}
void TC1_init(void)
{
  TCCR1A=0x81;   //COM1A1=1、COM1A0=0，比较匹配时清零 OC1A；WGM13=0、WGM12=1、
                 //WGM11=0，WGM10=1，选择 8 位快速 PWM 模式；CS12=0、CS11=1、
  TCCR1B=0x0A;          //CS10=0，选择 8 分频
   MCUCR=0x02;          //INT0 为下降沿时产生中断请求
    GICR=0x40;          //允许 INT0 产生中断
   #asm("sei")
}
void main(void)
{
    port_init();
    TC1_init();
    while(1)
      {
       if(a!=a_temp)
         {
          a_temp=a;
          switch(a)
            {
              case 0:
                OCR1A=00;
                count=00;
                break;
             case 1:
                OCR1A=25;
                count=10;
                break;
             case 2:
                OCR1A=51;
                count=20;
                break;
             case 3:
                OCR1A=76;
                count=30;
                break;
```

```
                case 4:
                  OCR1A=102;
                  count=40;
                  break;
                case 5:
                  OCR1A=127;
                  count=50;
                  break;
                case 6:
                  OCR1A=153;
                  count=60;
                  break;
                case 7:
                  OCR1A=178;
                  count=70;
                  break;
                case 8:
                  OCR1A=204;
                  count=80;
                  break;
                case 9:
                  OCR1A=229;
                  count=90;
                  break;
                case 10:
                  OCR1A=254;
                  count=99;
                  break;
                case 11:
                  count=0;
                  a=0;
                default:break;
               }
             }
           data_in();
           display();
         }
    }
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“用 TC1 的快速 PWM 模式输出占空比可调方波.COFF”文件，然后在 Proteus ISIS 中打开已创建的用 TC1 的快速 PWM 模式输出占空比可调方波电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮 ▶，进入调试状态。在调试状态下，可通过虚拟示波器观看 PD5 输出的 PWM 波形。单击按键 K 时，该方波的占空比发生改变，其仿真效果如图 7-23 所示。

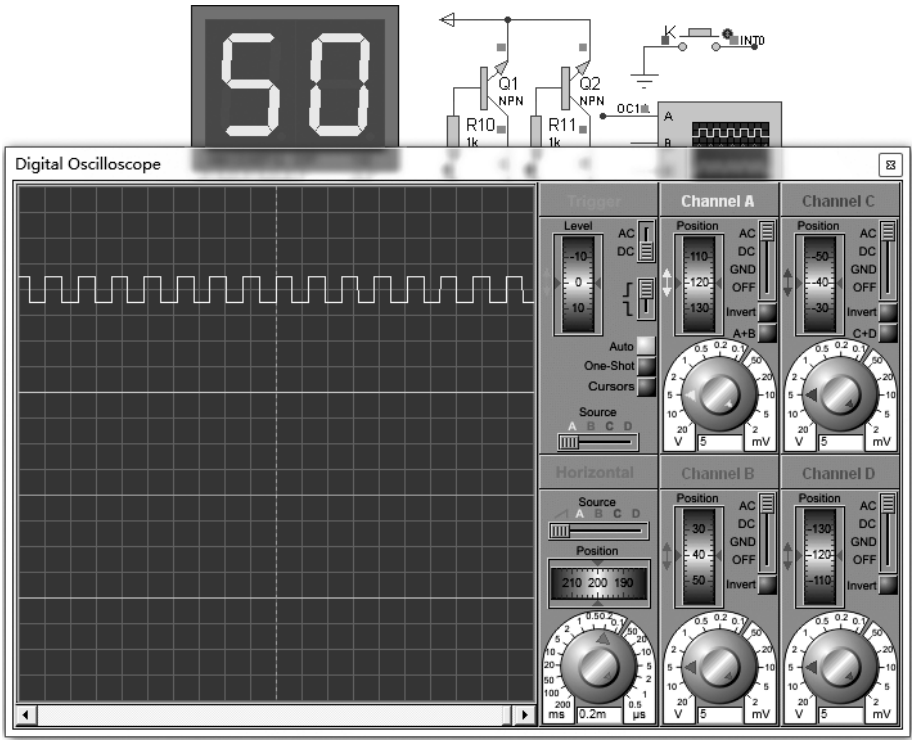


图 7-23　用 T/C1 的快速 PWM 模式输出占空比可调方波仿真效果图

任务 11　用 T/C1 的相位修改 PWM 模式输出占空比可调方波

使用 T/C1 的相位修正 PWM 模式设计一个占空比可调的 PWM 波形。要求单片机刚启动时，占空比为 0，以后每按一次按键，占空比以 10%递增，当占空比达到 90%，再按一次按键时，LED 显示为 99%。若再次按下按键后，又将占空比置为 0，如此循环。

硬件设计

本实例的电路与任务 10 中的图 7-21 完全一样，都是通过比较匹配输出端口 PD5（OC1A）来输出相应的 PWM 波形的。

软件设计

相位修正 PWM 模式可以产生高精度相位可调的 PWM 波形，更改输出比较寄存器的数值也可以改变输出波形的占空比。T/C1 有 8 位、9 位和 10 位 3 种分辨率的相位修正 PWM 模式。在 8 位分辨率的模式下，PWM 控制精度可以达到 1/256；在 9 位分辨率的模式下，PWM 控制精度可以达到 1/512；在 10 位分辨率的模式下，PWM 控制精度可以达到 1/1024。

本设计采用 10 位相位修正 PWM 模式，如果加载到输出比较寄存器的数值等于计数上

限值的 10%（OCR1A=102）时，就可以产生占空比为 10%的波形；如果等于上限值的 20%（OCR1A=204）时，就可以产生占空比为 20%的波形……其程序流程图请参见图 7-22，在此不再赘述。

源程序

```
/****************************************************
  文件名:          用 TC1 的相位修改 PWM 模式输出占空比可调方波.C
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define   CS1          PORTB.2              //与十位 LED 片选端连接
#define   CS2          PORTB.3              //与个位 LED 片选端连接
#define   button       PIND.2               //与按钮连接
#define   LED          PORTC
uchar count;
uchar a,a_temp;
uchar counth,countl;
flash uchar tab[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,  //共阳极 LED0～F 的段码
                   0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E};
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void data_in(void)
{
   counth=count/10;                        //十位分离
   countl=count%10;                        //个位分离
}
void display(void)                         //LED 数码管显示函数
 {
    LED=tab[counth];                       //送十位的段码值到 PORTC
    CS1=1;                                 //显示 LED 数码管的十位
    delay(5);                              //保持一定时间
    CS1=0;                                 //不显示 LED 数码管的十位
    LED=tab[countl];                       //送个位的段码值到 PORTC
    CS2=1;                                 //显示 LED 数码管的个位
    delay(5);                              //保持一定时间
    CS2=0;                                 //不显示 LED 数码管的个位
 }
```

```
interrupt [EXT_INT0] void int0_isr(void)       //INT0 中断函数
 {
    a++;
 }
void port_init(void)
 {
     DDRB=0xFF;
   PORTB=0xFF;
     DDRC=0xFF;
   PORTC=0xFF;
     DDRD=0xFB;
   PORTD=0xFF;
 }
void TC1_init(void)
{
  TCCR1A=0x83;   //COM1A1=1、COM1A0=0，比较匹配时清零 OC1A; WGM13=0、WGM12=1、
                 //WGM11=1，WGM10=1，选择 10 位快速 PWM 模式；CS12=0、CS11=1、
  TCCR1B=0x0A;  //CS10=0，选择 8 分频
   MCUCR=0x02;  //INT0 为下降沿时产生中断请求
    GICR=0x40;    //允许 INT0 产生中断
   #asm("sei")
}

 void main(void)
 {
     port_init();
     TC1_init();
     while(1)
        {
         if(a!=a_temp)
            {
             a_temp=a;
             switch(a)
               {
                  case 0:
                    OCR1A=00;
                    count=00;
                    break;
                case 1:
                    OCR1A=102;
                    count=10;
                    break;
                case 2:
                    OCR1A=204;
                    count=20;
                    break;
                case 3:
                    OCR1A=307;
```

```
                count=30;
                break;
            case 4:
                OCR1A=409;
                count=40;
                break;
            case 5:
                OCR1A=511;
                count=50;
                break;
            case 6:
                OCR1A=613;
                count=60;
                break;
            case 7:
                OCR1A=716;
                count=70;
                break;
            case 8:
                OCR1A=818;
                count=80;
                break;
            case 9:
                OCR1A=920;
                count=90;
                break;
            case 10:
                OCR1A=1022;
                count=99;
                break;
            case 11:
                count=0;
                a=0;
            default:break;
            }
        }
        data_in();
        display();
    }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“用 TC1 的相位修改 PWM 模式输出占空比可调方波.COFF”文件，然后在 Proteus ISIS 中打开已创建的用 TC1 的相位修改 PWM 模式输出占空比可调方波电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试状态下，可通过虚拟示波器观看 PD5 输出的 PWM 波

形。单击按键 K 时，该方波的占空比发生改变。

任务 12　用 T/C1 的输入捕捉模式测量低频脉冲周期

使用定时器/计数器 1 的输入捕捉模式测量外界输入的低频脉冲的周期。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 7-17 所列的元器件，并完成图 7-24 所示的硬件电路图设计。

表 7-17　用 T/C1 的输入捕捉模式测量低频脉冲周期所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	晶振 CRYSTAL 8MHz	电解电容 CAP-ELEC
电阻 RES	三极管 NPN	数码管 7SEG-MPX4-CA	555 多谐振荡器

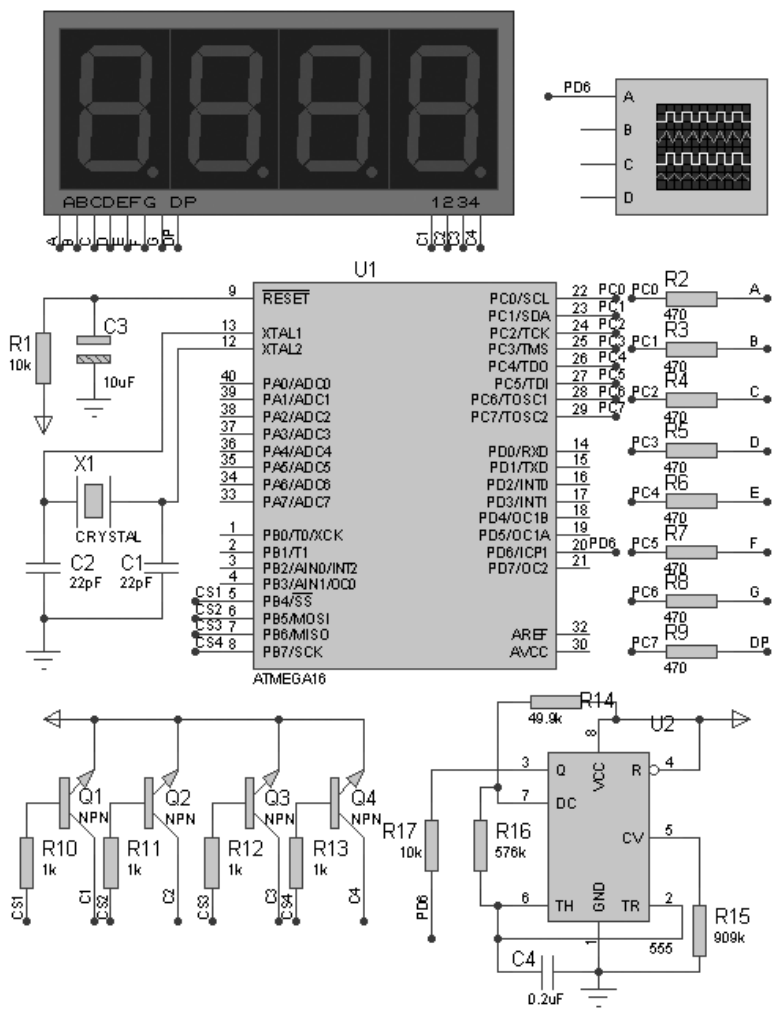


图 7-24　用 T/C1 的输入捕捉模式测量低频脉冲周期电路图

软件设计

T/C1 的输入捕捉功能可以捕捉一个外部事件的发生，记录事件发生的时间。当一个输入捕捉事件发生时（如 ICP1（PD6）上的信号发生上升沿跳变），T/C1 计数器 TCNT1 中的计数值被实时地写入输入捕捉寄存器 ICR1（ICR1H 和 ICR1L）中，并置位输入捕捉标志位 ICF1，产生中断申请。利用此特点，可以通过 ICP1 引脚测量外接低频脉冲的周期。其程序流程图如图 7-25 所示。

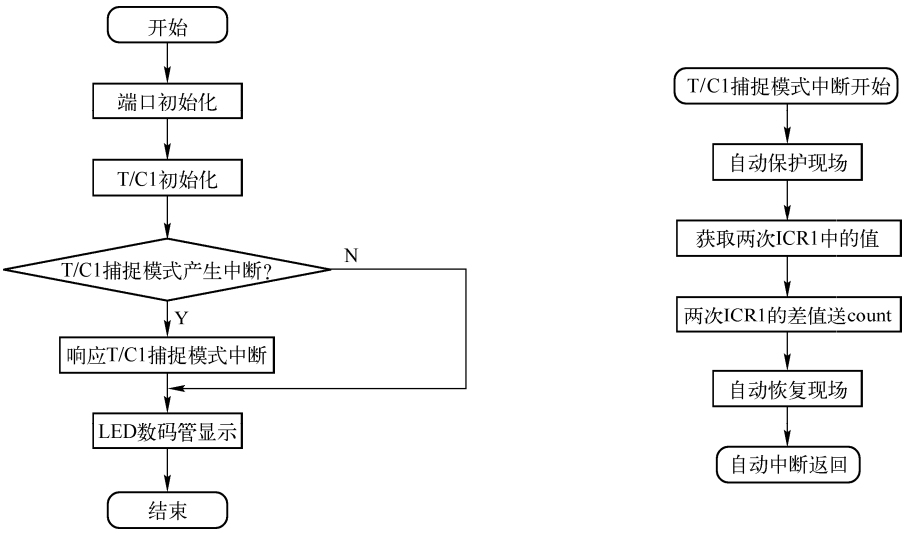


图 7-25　用 T/C1 的输入捕捉模式测量低频脉冲周期程序流程图

尽管采用输入捕捉功能测量外界脉冲周期的原理比较简单，但在实际应用中，由于系统需要 LED 扫描显示、频率值的换算也需要计算，且系统在运行过程中，还必须确保 T/C1 每次捕捉中断产生后马上将寄存器 ICR1H 和 ICR1L 中的时间读出，以及 T/C1 计数过程是否溢出等，所以采用此方式测量的周期存在一定的误差。

源程序

```
/****************************************************
  文件名:          用 TC1 的输入捕捉模式测量低频脉冲周期.C
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define   CS1          PORTB.4          //与千位 LED 片选端连接
#define   CS2          PORTB.5          //与百位 LED 片选端连接
#define   CS3          PORTB.6          //与十位 LED 片选端连接
#define   CS4          PORTB.7          //与个位 LED 片选端连接
#define   LED           PORTC
uchar count;
```

```
uchar a,a_temp;
uchar a_L,a_H;
uchar counthh,counthl,countlh,countll;
flash uchar tab[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,  //共阳极 LED0～F 的段码
                   0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E};
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void data_in(void)
{
    counthh=count/1000;                                  //千位分离
    counthl=(count-counthh*1000)/100;                    //百位分离
    countlh=(count-counthh*1000-counthl*100)/10;
    countll=count-counthh-counthl-countlh;
}
void display(void)                                       //LED 数码管显示函数
 {
    PORTB=0x0F;
    LED=tab[counthh];                                    //送千位的段码值到 PORTC
    CS1=1;                                               //显示 LED 数码管的千位
    delay(1);                                            //保持一定时间
    CS1=0;                                               //不显示 LED 数码管的千位
    LED=tab[counthl];                                    //送百位的段码值到
    CS2=1;                                               //显示 LED 数码管的百位
    delay(1);                                            //保持一定时间
    CS2=0;                                               //不显示 LED 数码管的百位
    LED=tab[countlh];                                    //送十位的段码值到 PORTC
    CS3=1;                                               //显示 LED 数码管的十位
    delay(1);                                            //保持一定时间
    CS3=0;                                               //不显示 LED 数码管的十位
    LED=tab[countll];                                    //送个位的段码值到 PORTC
    CS4=1;                                               //显示 LED 数码管的个位
    delay(1);                                            //保持一定时间
    CS4=0;                                               //不显示 LED 数码管的个位
 }
interrupt [TIM1_CAPT] void timer1_isr(void)
{
    a_temp=a;
    a_L=ICR1L;
    a_H=ICR1H;
```

```
        a=a_H*256+a_L;
        if(a<a_temp)
           {
            count=0xFFFF-a_temp+1+a;
           }
        else
           {
            count=a-a_temp;
           }
         TIFR=0x20;
}
void port_init()
{
      DDRB=0xFF;
     PORTB=0xFF;
      DDRC=0xFF;
     PORTC=0xFF;
      DDRD=0xBF;
     PORTD=0xFF;
}
void TC1_init(void)                          //T/C1 初始化设置
{
   TCCR1A=0x00;
   TCCR1B=0x41;
    TIMSK=0x20;
    #asm("sei")
}
void main(void)
{
   port_init();
   TC1_init();
     while(1)
       {
          data_in();
          display();
       }
 }
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“用 TC1 的输入捕捉模式测量低频脉冲周期.COFF”文件，然后在 Proteus ISIS 中打开已创建的用 TC1 的输入捕捉模式测量低频脉冲周期电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试状态下，可通过虚拟示波器观看 PD6 输出的 PWM 波形的周期时间，4 位共阳极数码管显示当前测量的周期时间，其仿真效果如图 7-26 所示。从图中可以看出，示波器测量的周期时间与数码管显示的周期时间确实存在一定的误差。

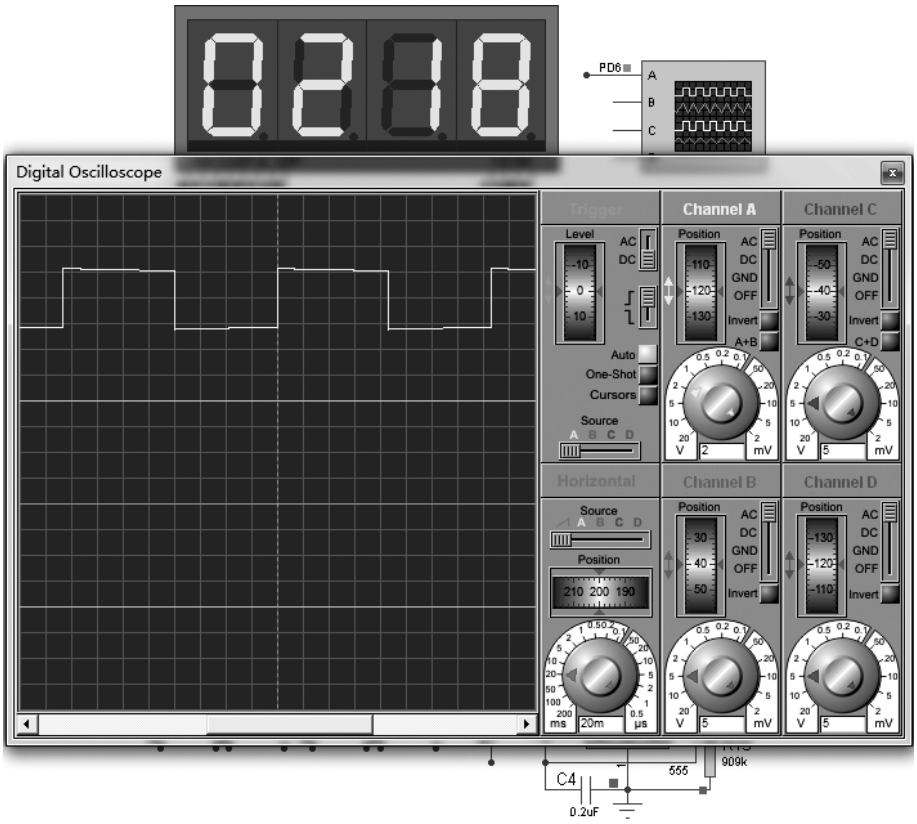


图 7-26　用 T/C1 的输入捕捉模式测量低频脉冲周期仿真效果图

任务 13　用 T/C2 实现简单门铃控制

假设单片机晶振频率为 8MHz，设计一个简单门铃控制系统。要求按下按键 K 时，蜂鸣器发出“叮咚”的声音，其声音由 T/C2 控制输出。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 7-18 所列的元器件，并完成图 7-27 所示的硬件电路图设计。

表 7-18　用 T/C2 实现简单门铃控制所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	晶振 CRYSTAL 8MHz	电解电容 CAP-ELEC
电阻 RES	按钮 BUTTON	蜂鸣器 SOUNDER	

软件设计

单片机控制蜂鸣器发出声音，实质上就是单片机控制某端口输出一个脉冲宽度不同的

高、低电平而已。脉冲宽度不同的高、低电平的产生也就是延时一段时间后持续输出高电平或低电平。

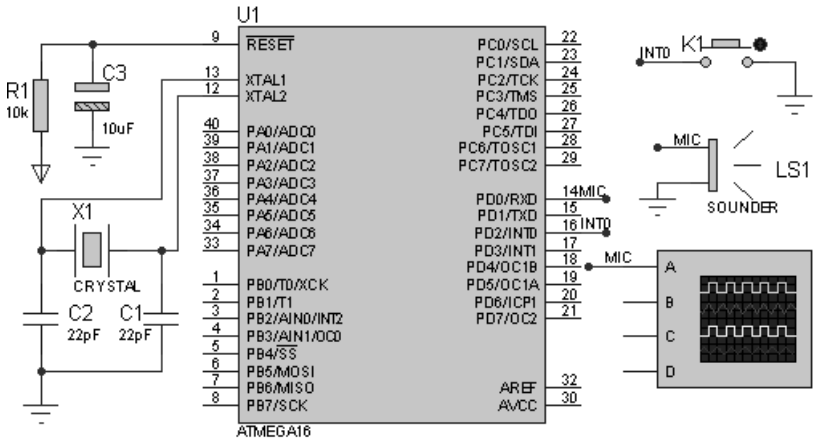


图 7-27　用 T/C2 实现简单门铃控制

在项目六的任务 4 中按下 K2 按键时，蜂鸣器中会发出“叮咚”的声音，只不过在程序中使用的是软件延时的方法。本任务虽然也是发出“叮咚”的声音，但是程序中使用硬件延时（T/C2 定时器中断）的方法。由于按键 K1 与单片机的 PD2 连接作为 INT0 的外部中断源，因此本任务采用了两种中断，即 INT0 中断和 T/C2 定时器中断，并且由 INT0 中断启动 T/C2 中断。其程序流程图如图 7-28 所示。

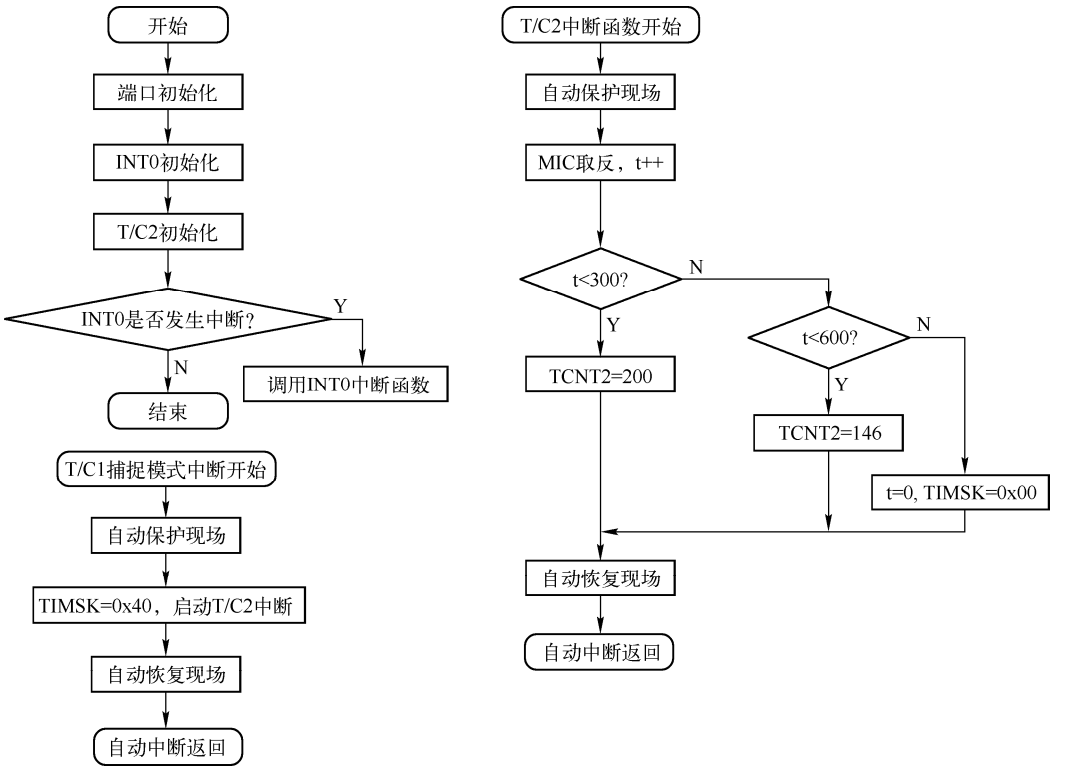


图 7-28　用 T/C2 实现简单门铃控制程序流程图

本设计中，通过 T/C2 产生“叮咚”的声音时，需要设置两个不同的时间，即 7ms 和 14ms，因此可使用 T/C2 的普通模式。假设采用的分频系数为 1024，系统时钟为 8MHz，根据公式 t=(255−TCNT2) × 分频系数/f_{osc} 来计算，得出 TCNT2=146 时，t=13.952ms；TCNT2=200 时，t=7.04ms。

采用中断方式进行延时，需编写 T/C2 的定时中断函数，在此函数中需将 200 和 146 重装 TCNT2。

由于定时器/计数器 2 的启动是在按下按键 K1 后才进行的，所以需编写 INT0 外部中断函数。如果按键 K1 被按下，则将 TOIE2 置为 1，即 TIMSK=0x40，启动 T/C2 溢出中断，使 T/C2 开始计时。

PROGRAM 源程序

```
/*****************************************************
  文件名:          用 TC2 实现简单门铃控制.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define   MIC              PORTD.0
#define   button1          PIND.2
uint t=0;
bit sp;
void delay(uint k)
{
   uint m,n;
   for(m=0;m<k;m++)
      {
         for(n=0;n<1140;n++)
           {;}
      }
}
interrupt [TIM2_OVF] void timer2_ovf_isr(void)
{
   sp=~sp;
   MIC=sp;
   t++;
   if(t<300)
      {
         TCNT2=200;                          //约 0.7ms
      }
   else if(t<600)
      {
         TCNT2=146;                          //约 1.4ms
      }
   else
```

```
        {
          t=0;
          TIMSK=0x00;                    //TOIE2=0，禁止 T/C2 溢出中断（暂停 T/C2 定时）
        }
}
interrupt[EXT_INT0] void int0_isr(void)  //INT0 中断函数
{
  delay(10);
  if(button1==0)                         //再次确认按键，软件防抖
    {
      TIMSK=0x40;                        //TOIE2=1，允许 T/C2 溢出中断（启动 T/C2 定时）
    }
}
void INT_init(void)                      //中断初始化设置
{
   MCUCR=0x00;                           //定义 INT0 为下降沿时产生中断
    GICR=0x40;                           //允许 INT0 产生中断
    #asm("sei")                          //开启总中断
}
void TC2_init(void)
{
   TCCR2=0x05;     //WGM21=0、WGM20=0，选择普通模式；COM21=0、COM20=0，使 PB3
          //为通用 I/O（OC2 不与引脚相连）；CS22=1、CS21=0、CS21=1，选择 1024 分频
   TCNT2=200;      //t=（255-TCNT2）×1024/8000000，t=0.005024，TCNT2=98
}
void port_init(void)                     //端口初始化
{
    DDRC=0xFF;
   PORTC=0xFF;
    DDRD=0xF3;
   PORTD=0xFF;
}
void main(void)
{
  port_init();
  INT_init();
  TC2_init();
  while(1)
    {
      ;
    }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“用 TC2 实现简单门铃控制.COFF”文件，然后在 Proteus ISIS 中打开已创建的用 TC2 实现简单门铃控制电路图并进行相应设置。

单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。未按下按键 K 时，蜂鸣器不发出任何声音，且虚拟示波器没有波形输出。按下按键 K 瞬间，虚拟示波器输出一个波形，之后输出的波形脉冲宽度发生了变化，而蜂鸣器发出清脆的“叮咚”声音。

任务 14　用 T/C2 的异步计数功能实现 59s 计时器

设计一个 59s 定时器，使用 T/C2 的异步计数功能作为硬件计时控制，要求还具有启动、暂停、复位功能，并用两位数码管进行时间显示。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 7-19 所列的元器件，并完成图 7-29 所示的硬件电路图设计。

表 7-19　用 T/C2 的异步计数功能实现 59s 计时器所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	晶振 X1:CRYSTAL 8MHz　X2:32.768kHz	电阻 RES
按钮 BUTTON	电解电容 CAP-ELEC	数码管 7SEG-MPX2-CA	三极管 NPN

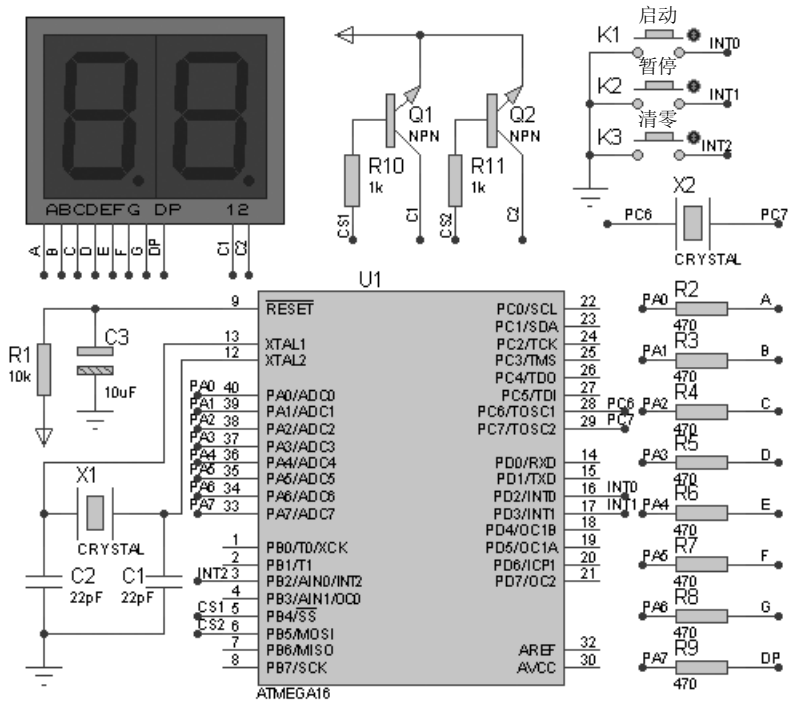


图 7-29　用 T/C2 的异步计数功能实现 59s 计时器电路图

软件设计

ATmega16 单片机的定时器/计数器 2 可以对外部异步时钟计数。外部的时钟源通过

TOSC1（PC6）和 TOSC2（PC7）的外接晶振获得。当外接晶振的频率为 32.768kHz，且计数源的时钟采用 128 分频时，计数器每计满 256 个数就可以产生一个溢出中断，时间间隔正好为 1s（128×256÷32768=1s），因此利用定时器/计数器 2 的这个特性，也可以进行 1s 的硬件延时。

由于定时器/计数器 2 的启动、停止、清零是在按下按键 K1、K2、K3 后才进行的，所以需编写 INT0、INT1 和 INT2 外部中断函数。在 INT0 外部中断函数中将 TOIE2 置为 1，即 TIMSK=0x40，启动 T/C2 溢出中断，使 T/C2 开始计时。在 INT0 外部中断函数中，则将 TOIE2 清零，即 TIMSK=0x00，禁止 T/C2 溢出中断，暂停 T/C2 计时。在 INT1 中断函数中，CPU 响应该中断（单击复位键）时，将显示数据 count 清零，且 TIMSK=0x00。

在主程序中，首先要对端口、定时器/计数器 2、外部中断 INT0、INT1 和 INT2 进行初始化，并启动总中断，然后调用 LED 七段数码管显示函数即可。定时器/计数器 2 的初始化主要是将 ASSR 置为 0x08，即启动计数功能；将 TCCR2 置为 0x05，即选择普通模式，采用 128 分频。其程序流程图如图 7-30 所示。

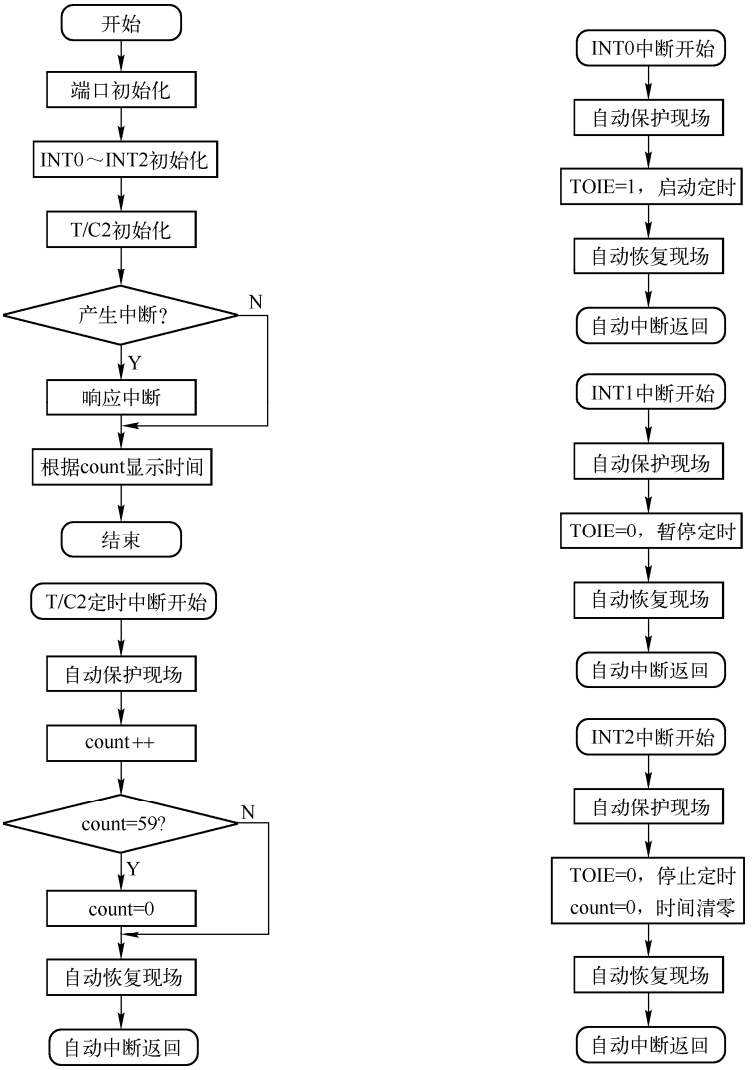


图 7-30　用 T/C2 的异步计数功能实现 59s 计时器程序流程图

PROGRAM 源程序

```
/*****************************************************
 文件名:          用 TC2 的异步计数功能实现 59s 计时器.c
 单片机型号:      ATmega16
 晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#define   uchar unsigned char
#define   uint unsigned int
#define   button1          PIND.2
#define   button2          PIND.3
#define   button3          PINB.2
#define   CS1               PORTB.4        //CS1 片选
#define   CS2               PORTB.5        //CS2 片选
#define   LED               PORTA
flash uchar tab[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,  //共阳极 LED0～F 的段码
                   0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E,0xBF,0x10};
uchar count;                                  //计数值
uchar data_H,data_L;                          //计数值高、低位
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
interrupt[EXT_INT0] void int0_isr(void)       //INT0 中断函数
{
  delay(10);
  if(button1==0)                              //再次确认按键，软件防抖
    {
      TIMSK=0x40;                             //TOIE2=1，允许 T/C2 溢出中断（启动 T/C2 定时）
    }
}
interrupt[EXT_INT1] void int1_isr(void)       //INT1 中断函数
{
  delay(10);
  if(button2==0)                              //再次确认按键，软件防抖
    {
      TIMSK=0x00;                             //TOIE2=0，禁止 T/C2 溢出中断（暂停 T/C2 定时）
    }
}
interrupt[EXT_INT2] void int2_isr(void)       //INT2 中断函数
{
```

```
    delay(10);
    if(button3==0)                          //再次确认按键，软件防抖
      {
        count=0;
        TIMSK=0x00;                         //TOIE2=0，禁止 T/C2 溢出中断
                                            //（暂停 T/C2 定时）
      }
}
interrupt [TIM2_OVF] void timer2_ovf_isr(void)   //T/C2 中断函数
{
    count++;
    if(count==59)
      {
        count=0;
      }
}
void data_in(void)                          //获取高、低位
{
   data_L=count%10;
   data_H=count/10;
}
void display(void)
{
   LED=tab[data_H];
   CS1=1;
   delay(5);
   CS1=0;
   LED=tab[data_L];
   CS2=1;
   delay(5);
   CS2=0;
}
void port_init(void)                        //端口初始化
{
     DDRA=0xFF;                             //PA 端口配置为输出
    PORTA=0xFF;                             //PA 端口初始值设置
     DDRB=0xFB;                             //PB 端口配置
    PORTB=0xFF;                             //PB 端口初始值设置
     DDRD=0xF3;                             //PD 端口配置为输出
    PORTD=0xFF;                             //PD 端口初始值设置
}
void INT_init(void)                         //中断初始化设置
{
    MCUCR=0x0A;                             //INT0、INT1 为下降沿时产生中断请求
   MCUCSR=0x00;                             //INT2 为下降沿时产生中断请求
     GICR=0xE0;                             //INT0、INT1、INT2 允许中断
     #asm("sei")                            //开启总中断
```

```
}
void TC2_init(void)
{
    ASSR=0x08;        //AS2=1，T/C2 由连接到 TOSC1 引脚的晶体振荡器驱动
    TCCR2=0x05;       //WGM21=0、WGM00=0，选择普通模式；COM21=0、COM20=0，正常的
                      //端口操作；CS22=1、CS21=0、CS21=1，选择 128 分频
    TCNT2=0x00;
}
void main(void)
{
  port_init();
  INT_init();
  TC2_init();
  while(1)
    {
      data_in();
      display();
    }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“用 TC2 的异步计数功能实现 59s 计时器.COFF”文件，然后在 Proteus ISIS 中打开已创建的用 TC2 的异步计数功能实现 59s 计时器电路图并进行相应设置。在 Proteus ISIS 编辑窗口中双击元器件 X2，在弹出的对话框中将频率改为 32.768kHz，如图 7-31 所示。

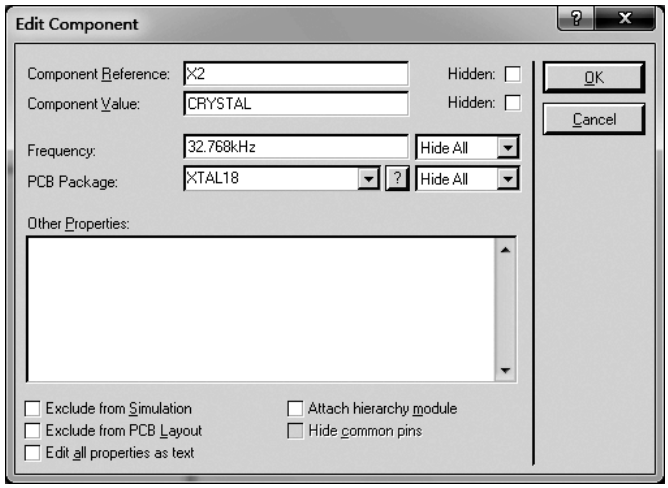


图 7-31　更改 X2 的频率

单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入虚拟仿真。从其仿真效果可以看出，它完全达到了所需的设计效果，如图 7-32 所示。

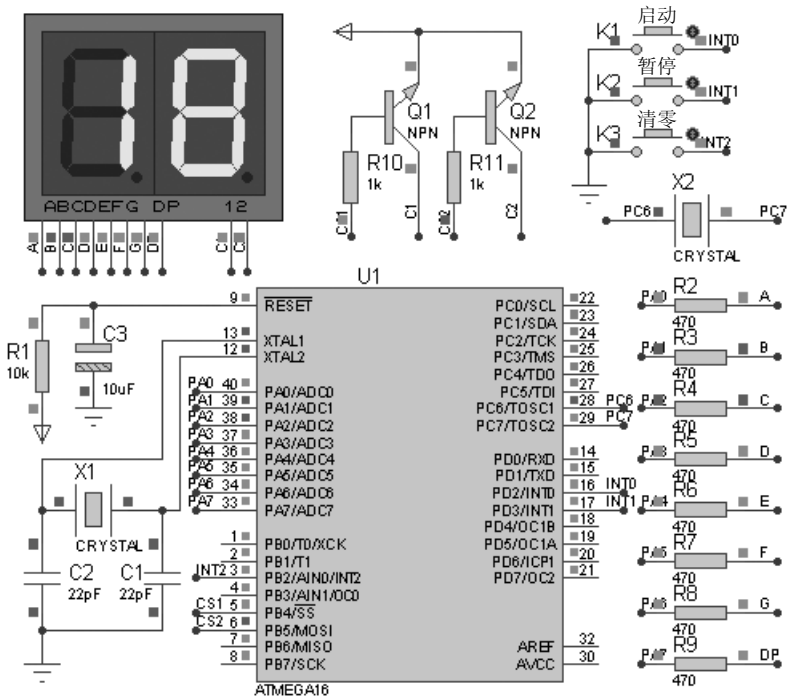


图 7-32　用 T/C2 的异步计数功能实现 59s 计时器仿真效果图

项目八　单片机串行通信设计

〖知识目标〗

☺ 了解 RS-232C 总线标准定义、ATmega16 单片机异步传输接口 USART、串行通信控制寄存器的定义、波特率的设置;

☺ 掌握串行通信的应用。

〖能力目标〗

☺ 学会串行中断相关寄存器的设置以及各方式下波特率计算;

☺ 学会单片机 USART 通信及其应用。

单片机系统设计中，经常要使用单片机串行口与外部设备进行通信，如单片机与单片机通信、单片机与 PC 通信等。本项目将介绍单片机串行通信的相关知识。



任务 1　单片机串行通信原理认知

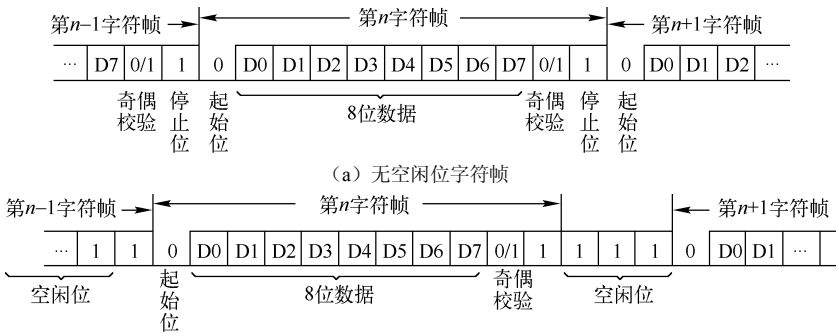
1．通信方式

按照串行数据的时钟控制方式，可将串行通信分为异步通信和同步通信两种方式。

1）**异步通信**　异步通信（Asynchronous Communication）中的数据是以字符（或字节）为单位组成字符帧（Character Frame）后进行传送的。这些字符帧在发送端逐帧地发送，在接收端通过数据线逐帧地接收字符或字节。在发送端和接收端，可以由各自的时钟控制数据的发送和接收，这两个时钟彼此独立，互不同步。

在异步串行数据通信中，有两个重要的指标，即字符帧和波特率。

【字符帧（Character Frame）】在异步串行数据通信中，字符帧也称为数据帧，它具有一定的格式，如图 8-1 所示。



（a）无空闲位字符帧

（b）有空闲位字符帧

图 8-1　串行异步通信字符帧格式

从图 8-1 中可以看出，字符帧由起始位、数据位、奇偶校验位、停止位等 4 部分组成。

☺ 起始位：位于字符帧的开头，只占一位，始终为逻辑低电平，发送器通过发送起始位表示一个字符传送的开始。

☺ 数据位：起始位之后紧跟着的是数据位。在数据位中规定，低位在前（左），高位在后（右）。由于字符编码方式不同，用户根据需要，数据位可取 5 位、6 位、7 位或 8 位。若传送的数据为 ASCII 字符，则数据位常取 7 位。

☺ 奇偶校验位：在数据位之后就是奇偶校验位，只占一位。用于检查传送字符的正确性。它有 3 种可能，即奇校验、偶校验或无校验，用户根据需要进行设定。

☺ 停止位：奇偶校验位之后为停止位。它位于字符帧的末尾，用于表示一个字符传送的结束，为逻辑高电平。通常停止位可取 1 位、1.5 位或 2 位，根据需要确定。

☺ 位时间：一个格式位的时间宽度。

☺ 帧（Frame）：从起始位开始到结束位为止的全部内容称为一帧。帧是一个字符的完整通信格式。因此也把串行通信的字符格式称为帧格式。

在串行通信中，发送端逐帧发送信息，接收端逐帧地接收信息，两相邻字符帧之间可以无空闲位，也可以有空闲位。图 8-1（a）所示为无空闲位，图 8-1（b）所示为 3 个空闲位的字符帧格式。两个相邻字符帧之间是否有空闲位由用户根据需要而决定。

【波特率（Band Rate）】数据传送的速率称为波特率，即每秒传送二进制代码的位数，也称为比特数，单位为 bit/s（bit per second，位/秒）。波特率是串行通信中的一个重要性能指标，用于表示数据传输的速度。波特率越高，数据传输速度越快。波特率和实际的字符传输速率不同，实际的字符传输速率是指每秒内所传字符帧的帧数，它和字符帧格式有关。

例如，波特率为 1200bit/s，若采用 10 个代码位的字符帧（1 个起始位，1 个停止位，8 个数据位），则字符的实际传送速率为 1200÷10=120 帧/秒；采用图 8-1（a）所示的字符帧，则实际的字符传送速率为 1200÷11=109.09 帧/秒；采用图 8-1（b）所示的字符帧，则实际的字符传送速率为 1200÷14=85.71 帧/秒;

每一位代码的传送时间 T_d 为波特率的倒数。例如波特率为 2400bit/s 的通信系统，每位的传送时间为

$$T_d = \frac{1}{2400} = 0.4167（\text{ms}）$$

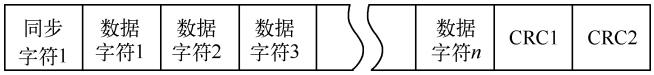
波特率与信道的频带有关，波特率越高，信道频带越宽。因此，波特率也是衡量通道频宽的重要指标。

在串行通信中，可以使用的标准波特率在 RS-232C 标准中已有规定，使用时应根据速度需要、线路质量等因素选定。

2）**同步通信**　同步通信（Synchronous Communication）是一种连续串行传送数据的通信方式，一次通信可传送若干个字符信息。同步通信的信息帧与异步通信的字符帧不同，它通常含有若干个数据字符，如图 8-2 所示。

图 8-2（a）所示为单同步字符帧结构，如图 8-2（b）所示为双同步字符帧结构。从图中可以看出，同步通信的字符帧由同步字符、数据字符、校验字符 CRC 三部分组成。同步字符位于字符帧的开头，用于确认数据字符的开始（接收端不断对传输线采样，并把采样的字符和双方约定的同步字符进行比较，比较成功后才把后面接收到的字符存储起来）；校验字符位于字符帧的末尾，用于对接收到的数据字符进行正确性的校验。数据字符长度由所需

传输的数据块长度决定。



（a）单同步字符帧结构



（b）双同步字符帧结构

图 8-2　串行同步通信的字符帧格式

在同步通信中，同步字符既可采用统一的标准格式，也可由用户约定。通常单同步字符帧中的同步字符采用的是 ASCII 码中规定的 SYN（即 16H）代码，双同步字符帧中的同步字符采用的是国际通用标准代码 EB90H。

同步通信的数据传输速率较高，通常可达 56kbit/s 或更高。但是同步通信要求发送时钟和接收时钟必须保持严格同步，且发送时钟频率应和接收时钟频率一致。

2．RS-232C 总线标准定义

RS-232C 是使用最早、应用最广的一种串行异步通信总线标准，是美国电子工业协会 EIA（Electronic Industry Association）的推荐标准。该标准定义了数据终端设备 DTE（Data Terminal Equipment）和数据通信设备 DCE（Data Communication Equipment）间按位串行传输的接口信息，合理安排了接口的电气信号和机械要求。DTE 是所传送数据的源或宿主，它可以是一台计算机、一个数据终端或一个外围设备；DCE 是一种数据通信设备，它可以是一台计算机或一个外围设备。例如，打印机与 CPU 之间的通信采用 RS-232C 接口。由于 STC89 系列单片机本身有一个全双工的串行接口，因此该系列的单片机可采用 RS-232C 接口标准与 PC 进行通信。

【RS-232C 引线功能】RS-232C 标准总线有 25 根和 9 根两种“D”形插头，25 芯插头座（DB-25）的引脚排列如图 8-3 所示，9 芯插头座（DB-9）的引脚排列如图 8-4 所示。

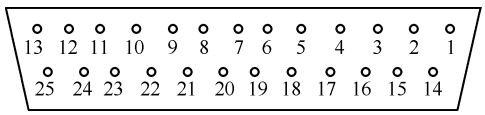


图 8-3　25 芯 RS-232C 引脚图

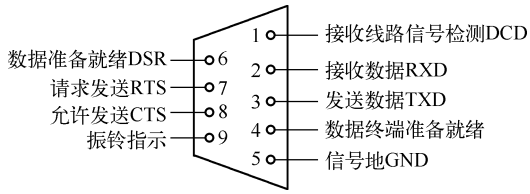


图 8-4　9 芯 RS-232C 引脚图

RS-232C 标准总线的 25 根信号线是为了各设备或器件之间进行联系或信息控制而定义的。各引脚的定义见表 8-1。

表 8-1　RS-232C 信号引脚定义

引　脚	名　称	定　　义	引　脚	名　称	定　　义
*1	GND	保护地	*3	RXD	接收数据
*2	TXD	发送数据	*4	RTS	请求发送

续表

引　脚	名　称	定　　义	引　脚	名　称	定　　义
*5	CTS	允许发送	16	SRXD	辅助通道接收数据
*6	DSR	数据准备就绪	17	RXC	接收时钟
*7	GND	信号地	18		未定义
*8	DCD	接收线路信号检测	19	SRTS	辅助通道请求发送
*9		接收线路建立检测	*20	DTR	数据终端准备就绪
10		线路建立检测	*21		信号质量检测
11		未定义	*22	RI	振铃指示
12	SDCD	辅助通道接收线信号检测	*23		数据信号速率选择
13	SCTS	辅助通道清除发送	*24		发送时钟
14	STXD	辅助通道发送数据	25		未定义
*15	TXC	发送时钟			

注：表中带“*”号的 15 根引线组成主信道通信，除了 11、18 及 25 三个引脚未定义外，其余的可作为辅信道进行通信，但是其传输速率比主信道要低，一般不使用。若使用，则主要用于传送通信线路两端所接的调制解调器的控制信号。

【RS-232C 接口电路】在微型计算机中，信号电平是 TTL 型的，即规定信号电压≥2.4V 时，为逻辑电平“1”；信号电压≤0.5V 时，为逻辑电平“0”。在串行通信中，若 DTE 和 DCE 之间采用 TTL 信号电平传送数据，如果两者的传送距离较大，很可能使源点的逻辑电平“1”在到达目的点时，就衰减到 0.5V 以下，使通信失败，所以 RS-232C 有其自己的电气标准。RS-232C 标准规定：在信号源点，信号电压在+5~+15V 时，为逻辑电平“0”，信号电压在–5~–15V 时，为逻辑电平“1”；在信号目的点，信号电压在+3~+15V 时，为逻辑电平“0”，信号电压在–3~–15V 时，为逻辑电平“1”，噪声容限为 2V。通常，RS-232C 总线电压为+12V 时表示逻辑电平“0”；–12V 时表示逻辑电平“1”。

由于 RS-232C 的电气标准不是 TTL 型的，在使用时不能直接与 TTL 型的设备相连，必须进行电平转换，否则会使 TTL 电路烧坏。

为实现电平转换，现通常采用电平转换芯片 MAX232 来完成，它是包含两路驱动器和接收器的 RS-232 转换芯片，如图 8-5 所示。从图中可以看出，MAX232 芯片内部有一个电压转换器，可以把输入的+5V 电压转换为 RS-232 接口所需的±10V 电压，尤其适用于没有±12V 的单电源系统。

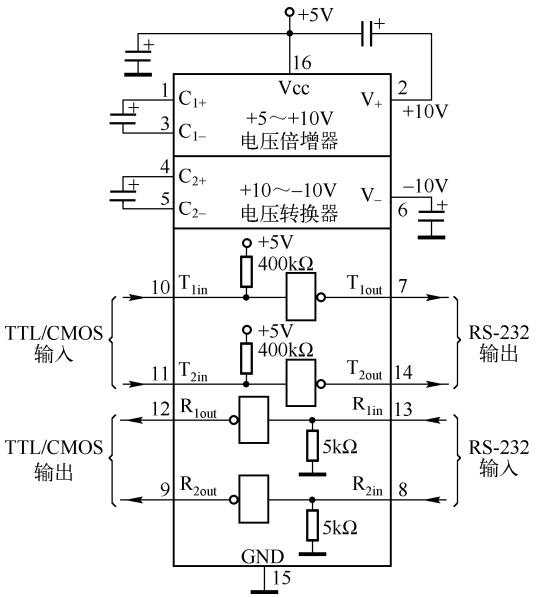


图 8-5　MAX232 的内部结构

3．异步传输接口 USART

ATmega16 单片机内部集成了一个全双工通用同步/异步串行数据收发器 USART，该

接口是一个高度灵活的串行通信设备，它由时钟发生器、数据发送器和数据接收器 3 部分构成。

时钟发生器包含同步逻辑，通过它可将波特率发生器及为从机同步操作所使用的外部输入时钟同步起来。XCK（发送器时钟）引脚只用于同步传输模式。发送器包括一个写缓冲器、串行移位寄存器、奇偶发生器及处理不同的帧格式所需的控制逻辑。使用写缓冲器，可以保持连续发送数据而不会在数据帧之间引入延迟。接收器具有时钟和数据恢复单元，它是 USART 模块中最复杂的部分。除了数据恢复单元外，接收器还包括奇偶位校验器、控制逻辑、移位寄存器的一个两级接收缓冲器（UDR）。接收器支持与发送器相同的帧格式，同时支持帧错误、数据溢出和奇偶校验错误的检测。

时钟发生器用于为发送器和接收器提供基本的时钟。USART 支持 4 种模式的时钟，即正常的异步模式、倍速的异步模式、主机同步模式和从机同步模式。USART 控制位 UMSEL 和状态寄存器 C（UCSRC）用于选择异步模式和同步模式。倍速模式（只适用于异步模式）受控于 UCSRA 寄存器的 U2X。使用同步模式（UMSEL=1）时，XCK 的数据方向寄存器（DDR_XCK）决定时钟源是由内部产生（主机模式）还是由外部产生（从机模式）的。XCK 引脚只在使用同步模式时有效。

【片内时钟产生（波特率发生器）】波特率发生器产生的时钟被用于异步模式与同步主机模式。USART 的波特率寄存器 UBRR 和预分频减 1 计数器相连接，一起构成可编程的预分频器或波特率发生器。减 1 计数器对系统时钟 OSC 计数，当其计数到零或 UBRRL 寄存器被写入时，会自动装入 UBRR 寄存器的数值。一旦计数到零时产生一个时钟，该时钟作为波特率发生器的输出时钟，输出时钟的频率为 f_{osc}/(UBRR+1)。发送器对波特率发生器的输出时钟可进行 2、8 或 16 的分频，具体情况取决于工作模式。波特率发生器的输出直接用于接收器与数据恢复单元。数据恢复单元使用了一个有 2、8 或 16 个状态的状态机。具体状态数由 UMSEL、U2X 与 DDR_XCK 位设定的工作模式决定。表 8-2 给出了计算波特率（bit/s）以及计算每一种使用内部时钟源工作模式的 UBRR 值的公式。

表 8-2　波特率计算公式

使 用 模 式	波特率的计算公式	UBRR 值的计算公式
异步正常模式（U2X=0）	$BAUD=\frac{f_{osc}}{16\times(UBRR+1)}$	$UBRR=\frac{f_{osc}}{16\times BAUD}-1$
异步倍速模式（U2X=1）	$BAUD=\frac{f_{osc}}{8\times(UBRR+1)}$	$UBRR=\frac{f_{osc}}{8\times BAUD}-1$
同步主机模式	$BAUD=\frac{f_{osc}}{2\times(UBRR+1)}$	$UBRR=\frac{f_{osc}}{2\times BAUD}-1$

注：表中 BAUD 为通信速率（bit/s）；f_{osc} 为系统时钟频率；UBRR 为波特率寄存器 UBRRH、UBRRL 中的值（0～4095）。

【倍速工作模式】通过设定 UCSRA 寄存器的 U2X 可以使传输速率加倍。该位只对异步工作模式有效。当工作在同步模式时，设置该位为“0”。

设置该位把波特率分频器的分频值从 16 降到 8，使异步通信的传输速率加倍。

> 注意：
> 在这种情况下，接收器只使用 50%的采样数对数据进行采样及时钟恢复，因此在该模式下需要更精确的系统时钟与更精确的波特率设置。发送器则没有这个要求。

【外部时钟】在同步从机操作模式中，数据的发送与接收是由外部时钟驱动的。输入到 XCK 引脚的外部时钟由同步寄存器进行采样，用以提高稳定性。同步寄存器的输出通过一个边沿检测器，然后应用于发送器与接收器。这一过程引入了 2 个 CPU 时钟周期的延时，因此外部 XCK 的最大时钟频率由下式限制：

$$f_{\text{xck}} < \frac{f_{\text{osc}}}{4}$$

使用同步模式时（UMSEL=1），XCK 引脚被用于时钟输入（从机模式）或时钟输出（主机模式）。时钟的边沿、数据的采样与数据的变化之间的关系的基本规律是，在改变数据输出端 TxD 的 XCK 时钟的相反边沿对数据输入端 RxD 进行采样。UCRSC 寄存器的 UCPOL 位用于选择使用 XCK 时钟的哪个边沿对数据进行采样和改变输出数据。当 UCPOL=0 时，在 XCK 的上升沿改变输出数据，在 XCK 的下降沿对 RxD 进行数据采样；当 UCPOL=1 时，在 XCK 的下降沿改变输出数据，在 XCK 的上升沿对 RxD 进行数据采样。

4．与串行接口相关的寄存器

【USART I/O 数据寄存器（UDR）】将数据写入 UDR 时，实际操作的是发送数据缓冲寄存器 TXB；读 UDR 时，实际返回的是接收数据缓冲寄存器 RXB 的内容。当设定使用 5、6 或 7 位的数字帧时，未用到的高位在发送时被忽略，在接收时由硬件自动清零。

Bit	7	6	5	4	3	2	1	0	
	RXB[7..0]								UDR（读）
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始值	0	0	0	0	0	0	0	0	
Bit	7	6	5	4	3	2	1	0	
	TXB[7..0]								UDR（写）
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始值	0	0	0	0	0	0	0	0	

只有当 UCSRA 寄存器的 UDRE 标志位置“1”后，才可以对发送缓冲器进行写操作。如果 UDRE 没有置“1”，则写入 UDR 的数据会被 USART 发送器忽略。当数据写入发送缓冲器后，若移位寄存器为空，发送器将把数据加载到发送移位寄存器，然后数据串行地从 TxD 引脚输出。

【USART 控制和状态寄存器 A（UCSRA）】

Bit	7	6	5	4	3	2	1	0
	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM
读/写	R	R/W	R	R	R	R	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ RXC（bit7）：USART 接收完成标志位。当收到的字符从接收移位寄存器传到 UDR 中（RXB）且未被读取时，该位置“1”。不论是否接收错误，该位都置“1”。当设置禁止接收时，UDR 中的数据就会刷新，同时 RXC 标志清零。RXC 的置位会产生接收完成的中断请求。RXC 在读 UDR 时自动清零。

☺ TXC（bit6）：USART 发送完成标志位。当发送移位寄存器的全部数据移出后，且在数据寄存器 UDR 中（TXB）没有待发送的数据时，该位置“1”。TXC 的置“1”会产生发送完成的中断请求，在转入执行完成中断时，TXC 由硬件自动清零。TXC 位也可以通过向

该位写一个逻辑“1”而清零。

☺ UDRE（bit5）：USART 数据寄存器空标志位。当写入 UDR（TXB）的字符被传送到发送寄存器时，该位置“1”，表示 UDR 可以写入新的发送数据。UDRE 的置“1”会产生发送数据寄存器空的中断请求。系统复位时，UDRE 置“1”，表示 USART 数据寄存器空，数据发送已准备好。

☺ FE（bit4）：接收帧出错标志位。如果接收缓冲器接收到的下一个字符有帧错误，即接收缓冲器中的下一个字符的第一个停止位为 0，则 FE 置“1”。该位一直保持为“1”，直到接收缓冲器（UDR）被读取为止。当接收到的停止位为 1 时，FE 标志为 0。对 UCSRA 进行写入时，这一位要写“0”。

☺ DOR（bit3）：接收数据溢出出错标志位。当接收缓冲器满（包含了两个数据），接收移位寄存器又有数据时，若检测到一个新的起始位，数据溢出就产生了。该位一直保持为“1”，直到接收缓冲器（UDR）被读取为止。对 UCSRA 进行重写时，这一位要设置为“0”。

☺ PE（bit2）：奇偶校验错误标志位。当奇偶校验使能（UPM1=1），且接收缓冲器中所接收到的下一个字符有奇偶校验错误时，PE 将置“1”。该位一直保持为“1”，直到接收缓冲 UDR 被读取为止。重写寄存器 UCSRA 的操作总是设置 PE 标志位为“0”。

☺ U2X（bit1）：USART 传输倍速控制位。该位只在异步模式下才有效，当使用同步模式时，应设置该位为“0”。如果设置该位为“1”，则将使波特率分频器的分频比由 16 降为 8，其效果是使异步通信速率提高 1 倍。

☺ MPCM（bit0）：多机通信模式允许控制位。该位使能多机通信模式，当 MPCM 位写为“1”时，所有接收到的数据帧（如果不包括地址信息）将被 USART 接收器忽略。USART 的发送模块不受 MPCM 设置的影响。

【USART 控制和状态寄存器 B（UCSRB）】

Bit	7	6	5	4	3	2	1	0
	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
初始值	0	0	0	0	0	0	0	0

☺ RXCIE（bit7）：接收结束中断使能位。该位置“1”后，使能 RXC 中断。当 RXCIE 为“1”，全局中断标志位 SREG 置位时，此时如果 UCSRA 寄存器的 RXC 也为“1”，则可以产生 USART 接收结束中断。

☺ TXCIE（bit6）：发送结束中断使能位。该位置“1”后，使能 TXC 中断。当 TXCIE 为“1”，全局中断标志位 SREG 置位时，此时如果 UCSRA 寄存器的 TXC 也为“1”，则可以产生 USART 发送结束中断。

☺ UDRIE（bit5）：USART 数据寄存器空中断使能位。当该位置“1”时，表示允许响应发送数据寄存器 UDR 空的中断请求。如果全局中断标志位 I 为“1”且 UDRIE 位为“1”，则当标志位 UDRE 置“1”时，一个发送数据寄存器空中断服务被执行。

☺ RXEN（bit4）：数据接收使能位。当该位置“1”时，允许 USART 接收数据。当接收器使能时，对应引脚的特性由通用数字 I/O 端口转变为 RXD。禁止数据接收时，将清除接收缓冲器中的数据，并使 FE、DOR、PE 标志位无效（为 0）。当接收数据被禁止后，USART 发送器将不再占用 RXD 引脚。

☺ TXEN（bit3）：数据发送使能位。当该位置“1”时，允许 USART 发送数据。当发送器使能时，对应引脚的特性由通用数字 I/O 端口转变为 TXD。当发送数据时禁止发送器（写 TXEN=0），则在移位寄存器中的数据和后续 UDR 中的数据被全部发送完成后，发送器才会禁止。当发送器禁止后，USART 发送器将不再占用 TXD 引脚。

☺ UCSZ2（bit2）：字符长度控制位。该位与 UCSRC 寄存器的 UCSZ1 和 UCSZ0 结合在一起，可以设置数据帧所包含的数据位数（字符长度）。

☺ RXB8（bit1）：接收数据的第 8 位。当采用接收的数据帧格式为 9 位数据帧时，RXB8 中接收到的是数据的第 9 数据位。读取 UDR 包含的低位数据前，需先读取 RXB8。

☺ TXB8（bit0）：发送数据的第 8 位。当采用发送数据帧格式为 9 位数据帧时，TXB8 中发送的是数据的第 9 数据位。写入 UDR 数据前，必须先写入 TXB8。

【USART 控制和状态寄存器 C（UCSRC）】

Bit	7	6	5	4	3	2	1	0
	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ URSEL（bit7）：寄存器选择控制位。由于 UCSRC 与 UBRRH 寄存器共用相同的 I/O 地址，所以需要用通过该位来选择到底是使用 UCSRC 寄存器还是 UBRRH 寄存器。写 UCSRC 寄存器时，该位必须写入“1”；读取 UCSRC 时，该位总是“1”。

☺ UMSEL（bit6）：USART 模式选择控制位。通过该位可选择 USART 为同步模式还是异步模式。当 UMSEL=0 时，USART 工作在异步模式；当 UMSEL=1 时，USART 工作在同步模式。

☺ UPM1、UPM0（bit5、bit4）：奇偶校验方式选择与使能位。如果已使能奇偶校验，则在发送数据时，发送器会自动产生并发送奇偶校验。对每一个接收到的数据，接收器都会产生一个奇偶值，并与 UPM0 所设置的值进行比较。如果不匹配，就将 UCSRA 中的 PE 置位，见表 8-3。

表 8-3　校验方式

UPM1	UPM0	校 验 方 式	UPM1	UPM0	校 验 方 式
0	0	无校验	1	0	使能偶校验
0	1	保留	1	1	使能奇校验

☺ USBS（bit3）：停止位的选择。该位用于选择发送数据帧中停止位的位数。当 USBS=0 时，为 1 位停止位；USBS=1 时，为 2 位停止位。对于接收器而言，这一位忽略。

☺ UCSZ1、UCSZ0（bit2、bit1）：字符长度控制位。这两位与 UCSRB 中的 UCSZ2 位结合在一起，可以设置数据帧包含的数据位（即字符长度），见表 8-4。

表 8-4　传送或接收的数据帧字符长度

UCSZ2	UCSZ1	UCSZ0	字符长度	UCSZ2	UCSZ1	UCSZ0	字符长度
0	0	0	5	1	0	0	保留
0	0	1	6	1	0	1	保留

续表

UCSZ2	UCSZ1	UCSZ0	字符长度	UCSZ2	UCSZ1	UCSZ0	字符长度
0	1	0	7	1	1	0	保留
0	1	1	8	1	1	1	9

☺ UCPOL（bit0）：时钟极性的选择。该位只能在同步模式下使用。在异步模式下，将这一位清零。UCPOL 设置了串行输出数据的改变和输入数据采样与同步时钟 XCK 之间的关系，见表 8-5。

表 8-5　时钟极性的选择

UCPOL	发送数据的改变（TXD 引脚输出）	接收数据的改变（RXD 引脚输入）
0	XCK 上升沿	XCK 下降沿
1	XCK 下降沿	XCK 上升沿

【波特率寄存器（UBRRL 和 UBRRH）】

Bit	15	14	13	12	11	10	9	8	
	URSEL	-	-	-	UBRR[11..8]				UBRRH
读/写	R/W	R	R	R	R/W	R/W	R/W	R/W	
初始值	0	0	0	0	0	0	0	0	
Bit	7	6	5	4	3	2	1	0	
	UBRR[7..0]								UBRRL
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始值	0	0	0	0	0	0	0	0	

☺ URSEL（bit15）：寄存器选择控制位。由于 UCSRC 与 UBRRH 寄存器共用相同的 I/O 地址，所以需用通过该位来选择到底是使用 UCSRC 寄存器还是 UBRRH 寄存器。写 UBRRH 寄存器时，该位必须写入“0”；读取 UBRRH 时，该位总是“0”。

☺ bit14~bit12：保留位。这些位是为以后的使用而保留的。为了与以后的器件兼容，写 UBRRH 时应将这些位清零。

☺ UBRR（bit11~bit0）：USART 波特率设置寄存器。由 UBRRH 低 4 位和寄存器 UBRRL 的 8 位构成一个 12 位的寄存器，用于对 USART 传送或接收波特率进行设置。如果波特率设置被改变，则正在进行的数据传输将受到破坏。写 UBRRL 将立即更新波特率预分频器的设置。

5．波特率的设置

对于标准频率的晶体，大多数异步操作的波特率设置（UBRR 数值）可以从表 8-6～表 8-9 中获得。

使用表中的设置值产生的时钟波特率与实际的波特率的偏差小于±0.5%。虽然更高的波特率偏差也可以使用，但会降低接收器的抗干扰性。误差可以通过以下公式计算：

$$\text{误差（\%）}=\left(\frac{\text{BaudRate}_{\text{ClosetMatch}}}{\text{BaudRate}}\right)\times 100\%$$

表 8-6　不同晶振频率的 UBRR 设置（1）

波特率（bit/s）	f_{osc}=1.0000MHz				f_{osc}=1.8432MHz				f_{osc}=2.0000MHz			
	U2X=0		U2X=1		U2X=0		U2X=1		U2X=0		U2X=1	
	UBRR	误差（%）	UBRR	误差（%）	UBRR	误差（%）	UBRR	误差（%）	UBRR	误差（%）	UBRR	误差（%）
2400	25	0.2	51	0.2	47	0.0	95	0.0	51	0.2	103	0.2
4800	12	0.2	25	0.2	23	0.0	47	0.0	25	0.2	51	0.2
9600	6	–7.0	12	0.2	11	0.0	23	0.0	12	0.2	25	0.2
14.4k	3	8.5	8	–3.5	7	0.0	15	0.0	8	–3.5	16	2.1
19.4k	2	8.5	8	–7.0	5	0.0	11	0.0	6	–7.0	12	0.2
28.8k	1	8.5	3	8.5	3	0.0	7	0.0	3	8.5	8	–3.5
38.4k	1	–18.6	2	8.5	2	0.0	5	0.0	2	8.5	6	–7.0
57.6k	0	8.5	1	8.5	1	0.0	3	0.0	1	8.5	3	8.5
76.8k	–	–	1	–18.6	1	–25.0	2	0.0	1	–18.6	2	8.5
115.2k	–	–	0	8.5	0	0.0	1	0.0	0	8.5	1	8.5
230.4k	–	–	–	–	–	–	0	0.0	–	–	–	–
250k	–	–	–	–	–	–	–	–	–	–	0	0.0
最大[注]	62.5kbit/s		125kbit/s		115.2kbit/s		230.4kbit/s		125kbit/s		250kbit/s	

注：UBRR=0，误差=0.0%，表 8-7~表 8-9 类同。

表 8-7　不同晶振频率的 UBRR 设置（2）

波特率（bit/s）	f_{osc}=3.6864MHz				f_{osc}=4.0000MHz				f_{osc}=7.3728MHz			
	U2X=0		U2X=1		U2X=0		U2X=1		U2X=0		U2X=1	
	UBRR	误差（%）	UBRR	误差（%）	UBRR	误差（%）	UBRR	误差（%）	UBRR	误差（%）	UBRR	误差（%）
2400	95	0.0	191	0.0	103	0.2	207	0.2	191	0.0	383	0.0
4800	47	0.0	95	0.0	51	0.2	103	0.2	95	0.0	191	0.0
9600	23	0.0	47	0.0	25	0.2	51	0.2	47	0.0	95	0.0
14.4k	15	0.0	31	0.0	16	2.1	34	–0.8	31	0.0	63	0.0
19.4k	11	0.0	23	0.0	12	0.2	25	0.2	23	0.0	47	0.0
28.8k	7	0.0	15	0.0	8	–3.5	16	2.1	15	0.0	31	0.0
38.4k	5	0.0	11	0.0	6	–7.0	12	0.2	11	0.0	23	0.0
57.6k	3	0.0	7	0.0	3	8.5	8	–3.5	7	0.0	15	0.0
76.8k	2	0.0	5	0.0	2	8.5	6	–7.0	5	0.0	11	0.0
115.2k	1	0.0	3	0.0	1	8.5	3	8.5	3	0.0	7	0.0
230.4k	0	0.0	1	0.0	0	8.5	1	8.5	1	0.0	3	0.0
250k	0	–7.8	1	–7.8	0	0.0	1	0.0	1	–7.8	3	–7.8
0.5M	–	–	0	–7.8	–	–	0	0.0	0	–7.8	1	–7.8
1M	–	–	–	–	–	–	–	–	–	–	0	–7.8
最大[注]	230.4kbit/s		460.8kbit/s		250kbit/s		500kbit/s		460.8kbit/s		921.6kbit/s	

表 8-8　不同晶振频率的 UBRR 设置（3）

波特率（bit/s）	f_{osc}=8.0000MHz				f_{osc}=11.0592MHz				f_{osc}=14.7456MHz			
	U2X=0		U2X=1		U2X=0		U2X=1		U2X=0		U2X=1	
	UBRR	误差（%）	UBRR	误差（%）	UBRR	误差（%）	UBRR	误差（%）	UBRR	误差（%）	UBRR	误差（%）
2400	207	0.2	416	–0.1	287	0.0	575	0.0	383	0.0	767	0.0
4800	103	0.2	207	0.2	143	0.0	287	0.0	191	0.0	383	0.0
9600	51	0.2	103	0.2	71	0.0	143	0.0	95	0.0	191	0.0
14.4k	34	–0.8	68	0.6	47	0.0	95	0.0	63	0.0	127	0.0
19.4k	25	0.2	51	0.2	35	0.0	71	0.0	47	0.0	95	0.0
28.8k	16	2.1	34	–0.8	23	0.0	47	0.0	31	0.0	63	0.0
38.4k	12	0.2	25	0.2	17	0.0	35	0.0	23	0.0	47	0.0
57.6k	8	–3.5	16	2.1	11	0.0	23	0.0	15	0.0	31	0.0
76.8k	6	–7.0	12	0.2	8	0.0	17	0.0	11	0.0	23	0.0
115.2k	3	8.5	8	–3.5	5	0.0	11	0.0	7	0.0	15	0.0
230.4k	1	8.5	3	8.5	2	0.0	5	0.0	3	0.0	7	0.0
250k	1	0.0	3	0.0	2	–7.8	5	–7.8	3	–7.8	6	5.3
0.5M	0	0.0	1	0.0	–	–	2	–7.8	1	–7.8	3	–7.8
1M	–	–	0	0.0	–	–	–	–	0	–7.8	1	–7.8
最大[注]	500kbit/s		1Mbit/s		691.2kbit/s		1.3824Mbit/s		921.6kbit/s		1.8432Mbit/s	

表 8-9　不同晶振频率的 UBRR 设置（4）

波特率（bit/s）	f_{osc}=16.0000MHz				f_{osc}=18.4320MHz				f_{osc}=20.0000MHz			
	U2X=0		U2X=1		U2X=0		U2X=1		U2X=0		U2X=1	
	UBRR	误差（%）	UBRR	误差（%）	UBRR	误差（%）	UBRR	误差（%）	UBRR	误差（%）	UBRR	误差（%）
2400	416	–0.1	832	0.0	479	0.0	959	0.0	520	0.0	1041	0.0
4800	207	0.2	416	–0.1	239	0.0	479	0.0	259	0.2	520	0.0
9600	103	0.2	207	0.2	119	0.0	239	0.0	129	0.2	259	0.2
14.4k	68	0.6	138	–0.1	79	0.0	159	0.0	86	–0.2	173	–0.2
19.4k	51	0.2	103	0.2	59	0.0	119	0.0	64	0.2	129	0.2
28.8k	34	–0.8	68	0.6	39	0.0	79	0.0	42	0.9	86	–0.2
38.4k	25	0.2	51	0.2	29	0.0	59	0.0	32	–1.4	64	0.2
57.6k	16	2.1	34	–0.8	19	0.0	39	0.0	21	–1.4	42	0.9
76.8k	12	0.2	25	0.2	14	0.0	29	0.0	15	1.7	32	–1.4
115.2k	8	–3.5	16	2.1	9	0.0	19	0.0	10	–1.4	21	–1.4
230.4k	3	8.5	8	–3.5	4	0.0	9	0.0	4	8.5	10	–1.4
250k	3	0.0	7	0.0	4	–7.8	8	2.4	4	0.0	9	0.0
0.5M	1	0.0	3	0.0	–	–	4	–7.8	–	–	44	0.0
1M	0	0.0	1	0.0	–	–	–	–	–	–	–	–
最大[注]	1Mbit/s		2Mbit/s		1.152Mbit/s		230.4kbit/s		125kbit/s		250kbit/s	

任务 2　甲机通过串口控制乙机 LED 显示状态

使用两个单片机进行 USART 通信，要求一个单片机 U1（甲机）作为数据发送机，另一个单片机 U2（乙机）作为数据接收机。发送机的 PB 端口外接 8 个拨动开关；接收机外接 8 个 LED。8 个 LED 对应 8 个拨动开关的状态，即发送机将 PB 端口获得的状态值通过 TXD 发送给接收机，接收机先通过 RXD 接收数据，再将该数据通过 PC 端口的 LED 显示出来。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 8-10 所列的元器件，并完成图 8-6 所示的硬件电路图设计。

表 8-10　甲机通过串口控制乙机 LED 显示状态所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	晶振 CRYSTAL 8MHz	电解电容 CAP-ELEC
电阻 RES	DB-9　CONN-D9F	发光二极管 LED-BLUE	发光二极管 LED-GREEN
电平转换 MAX232	拨码开关 DIPSW_8	发光二极管 LED-RED	发光二极管 LED-YELLOW

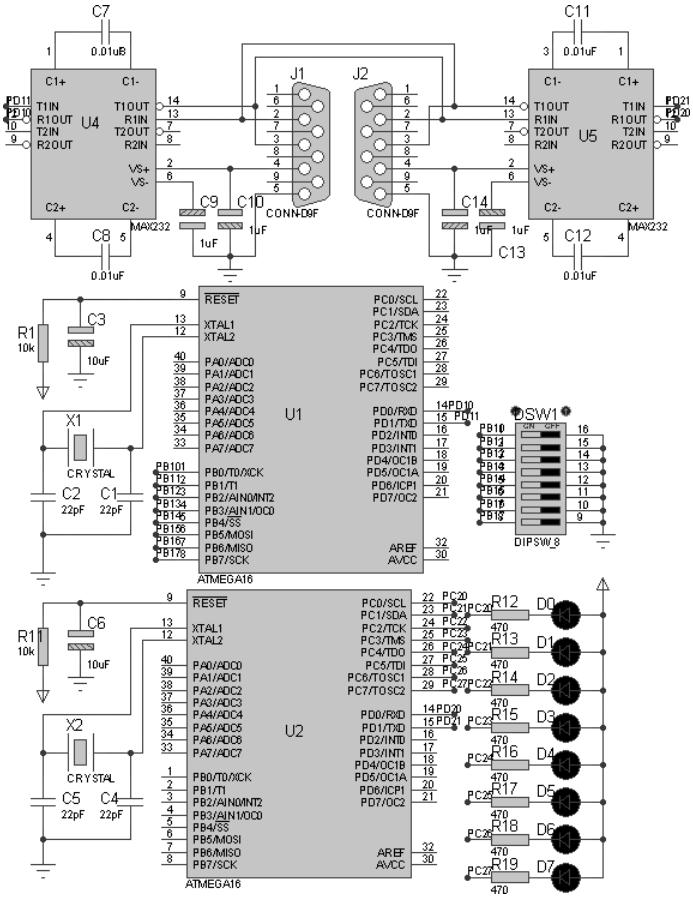


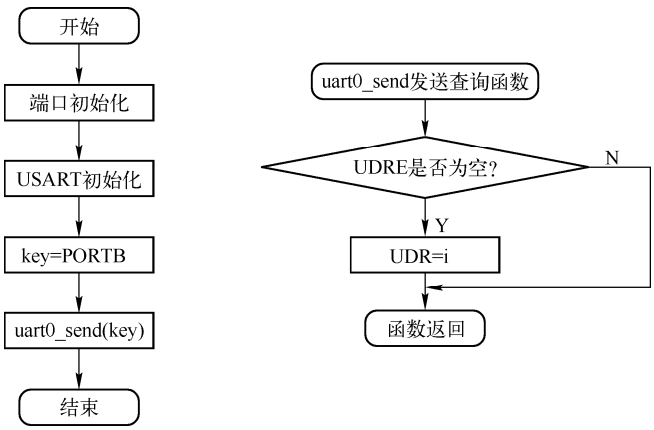
图 8-6　甲机通过串口控制乙机 LED 显示状态电路图

软件设计

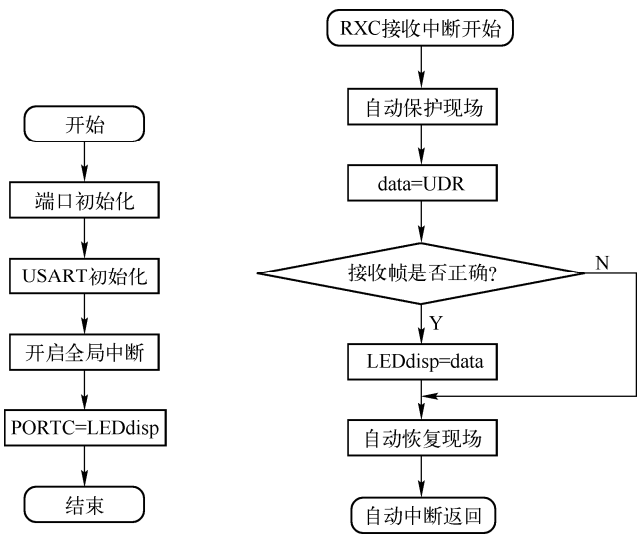
本任务属于单片机双机通信，这两个单片机都工作在半工状态，即甲机只负责将拨码开关的状态发送出去，乙机只负责将开关状态接收过来并进行显示。每个单片机均需要一个源程序，因此该设计应有两个程序，即发送源程序和接收源程序。

在发送源程序中，编写一个发送查询函数。在此函数中主要判断 USART 的 UDRE 标志是否为空，若为空，则将 i 送入 UDR。采用这种方法，首先在主程序中调用该函数，然后将 key（key 值就是 PB 端口的状态值）送给 i 即可完成键值的传送。其程序流程图如图 8-7（a）所示。

在接收源程序中，编写一个 RXC 接收中断函数。在此函数中先将接收的数据 UDR 送入 data，再通过 UCSRA 判断在传输中是否发生帧错误或有数据溢出或发生奇偶校验错误，如果未发生这些错误，则将 data 送给 LEDdisp，使该数据在主程序中通过 PC 端口的 LED 管显示出来。其程序流程图如图 8-7（b）所示。



（a）U1 发送程序流程图



（b）U2 接收程序流程图

图 8-7　甲机通过串口控制乙机 LED 显示状态程序流程图

源程序

【U1 发送源程序】

```
/*****************************************************
 文件名:          U1 发送程序.C
 单片机型号:      ATmega16
 晶振频率:        8.0MHz
 说明:            U1 机将开关状态发送给 U2 机
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define RXC   7
#define UDRE 5
uchar key;
void uart0_send(uchar i)                  //数据发送
{
    while(!(UCSRA&(1<<UDRE)));
    UDR=i;
}
void USART_init(void)                     //USART 初始化
{
    UCSRA=0x00;
    UBRRH=0;
    UBRRL=25;                             //系统时钟为 8MHz，波特率为 9600bit/s
    UCSRB=0x18;
    UCSRC=0x86;
}
void port_init(void)                      //端口初始化
{
     DDRB=0x00;
    PORTB=0xFF;
}
void main(void)
 {
    port_init();
    USART_init();
    while(1)
       {
          key=PINB;
          uart0_send(key);
       }
 }
```

【U2 接收源程序】

```
/*****************************************************
```

```
 文件名:          U2 接收程序.C
 单片机型号:      ATmega16
 晶振频率:        8.0MHz
 说明:            U2 机显示 U1 机开关状态
*************************************************/
#include "mega16.h"
#define   uchar unsigned char
#define   uint unsigned   int
#define   LED              PORTC
#define   UDRE              5
#define   FE               4
#define   PE               2
#define   DOR               3
uchar LEDdisp;
interrupt [USART_RXC] void usart_rx_isr(void)       //USART 串行接收中断
{
    uchar status,data;
    status=UCSRA;
    data=UDR;
    if((status&((1<<FE)|(1<<PE)|(1<<DOR)))==0)
     {
       LEDdisp=data;
     }
}
void port_init(void)
{
    DDRC=0xFF;
    PORTC=0xFF;
}
void USART_init(void)
{
    UCSRA=0x00;
    UBRRH=0;
    UBRRL=25;                                       //系统时钟为 8MHz，波特率为 9600bit/s
    UCSRB=0x90;
    UCSRC=0x86;
    #asm("sei")
}
void main(void)
 {
    port_init();
    USART_init();
    while(1)
     {
       LED=LEDdisp;
     }
 }
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“U1 发送程序.COFF”和“U2 接收程序.COFF”文件，然后在 Proteus ISIS 中打开已创建的甲机通过串口控制乙机 LED 显示状态电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入程序仿真。改变与 U1 相连的 DIPSW_8 开关位置，与 U2 连接的相应的 LED 点亮或熄灭，仿真效果如图 8-8 所示。

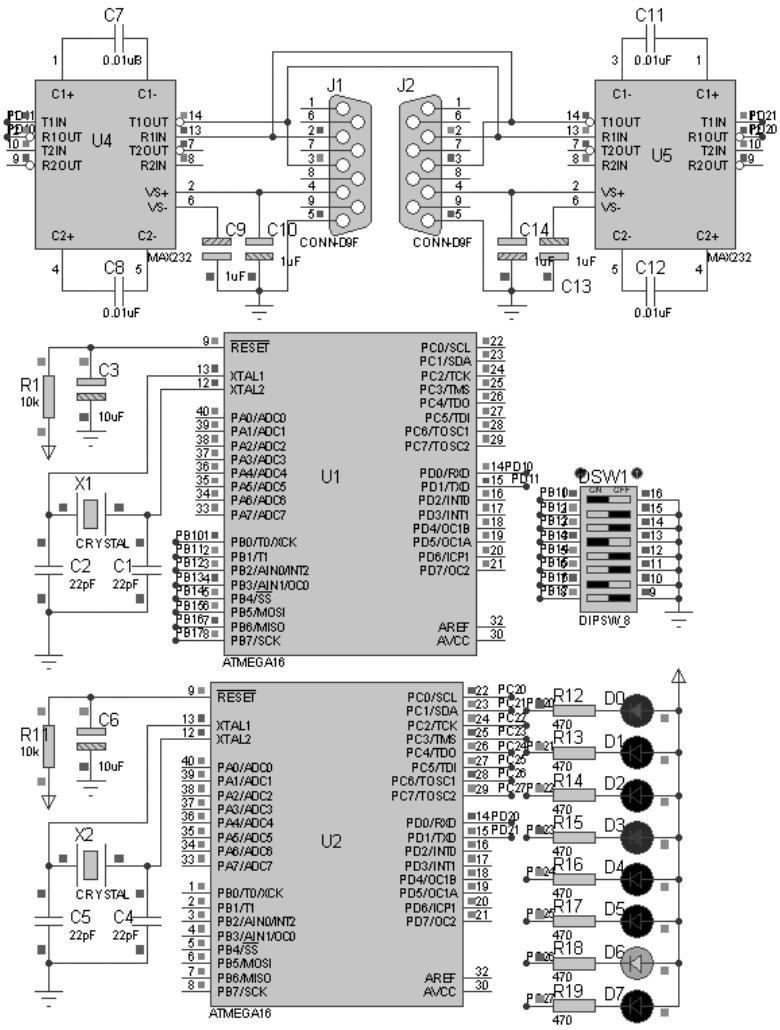


图 8-8　甲机通过串口控制乙机 LED 显示状态仿真效果图

任务 3　甲机通过串口控制乙机计时

单片机甲机（U1）的 PD2 外接按键 K1，PD3 外接按键 K2，PD2 外接按键 K3；单片机乙机（U2）PC、PB4、PB5 端口作为输出口，外接一个 2 位 LED 数码管。要求使用单片

机串行通信，使甲机的按键 K1 作为乙机计时的“开始”控制按键，甲机的按键 K2 作为乙机计时的“暂停”控制按键，甲机的按键 K3 作为乙机计时的“复位”控制按键。乙机的计时由其 T/C0 来实现。



硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 8-11 所列的元器件，并完成如图 8-9 所示的硬件电路图设计。

表 8-11　甲机通过串口控制乙机计时所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	晶振 CRYSTAL 8MHz	电解电容 CAP-ELEC
电阻 RES	按钮 BUTTON	数码管 7SEG-MPX2-CA	三极管 NPN
电平转换 MAX232	DB-9　CONN-D9F		

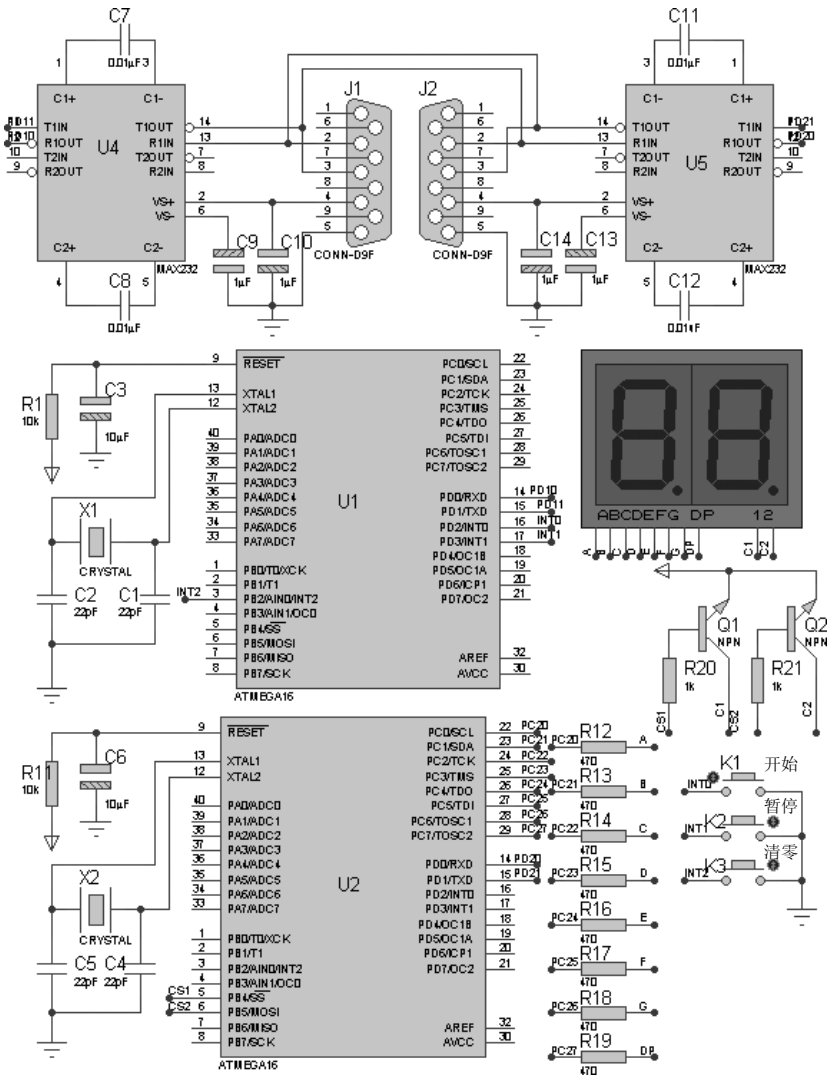
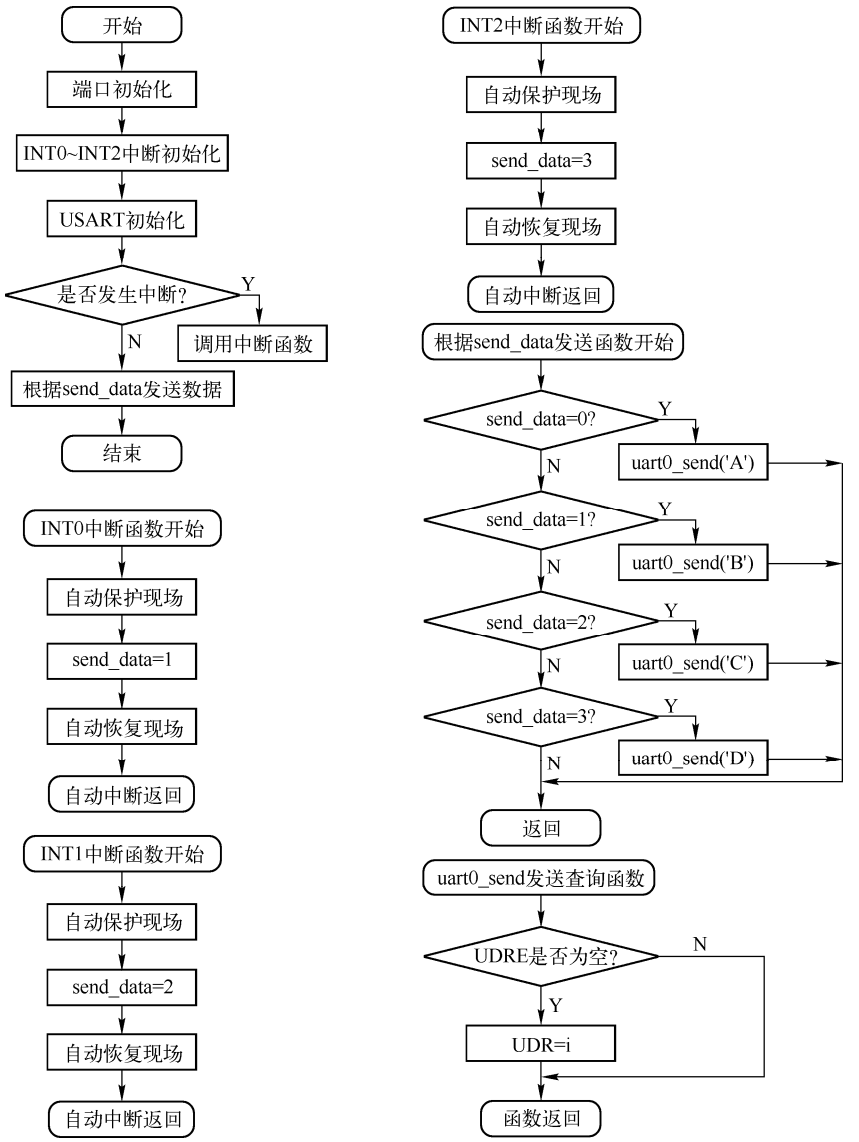


图 8-9　甲机通过串口控制乙机计时电路图

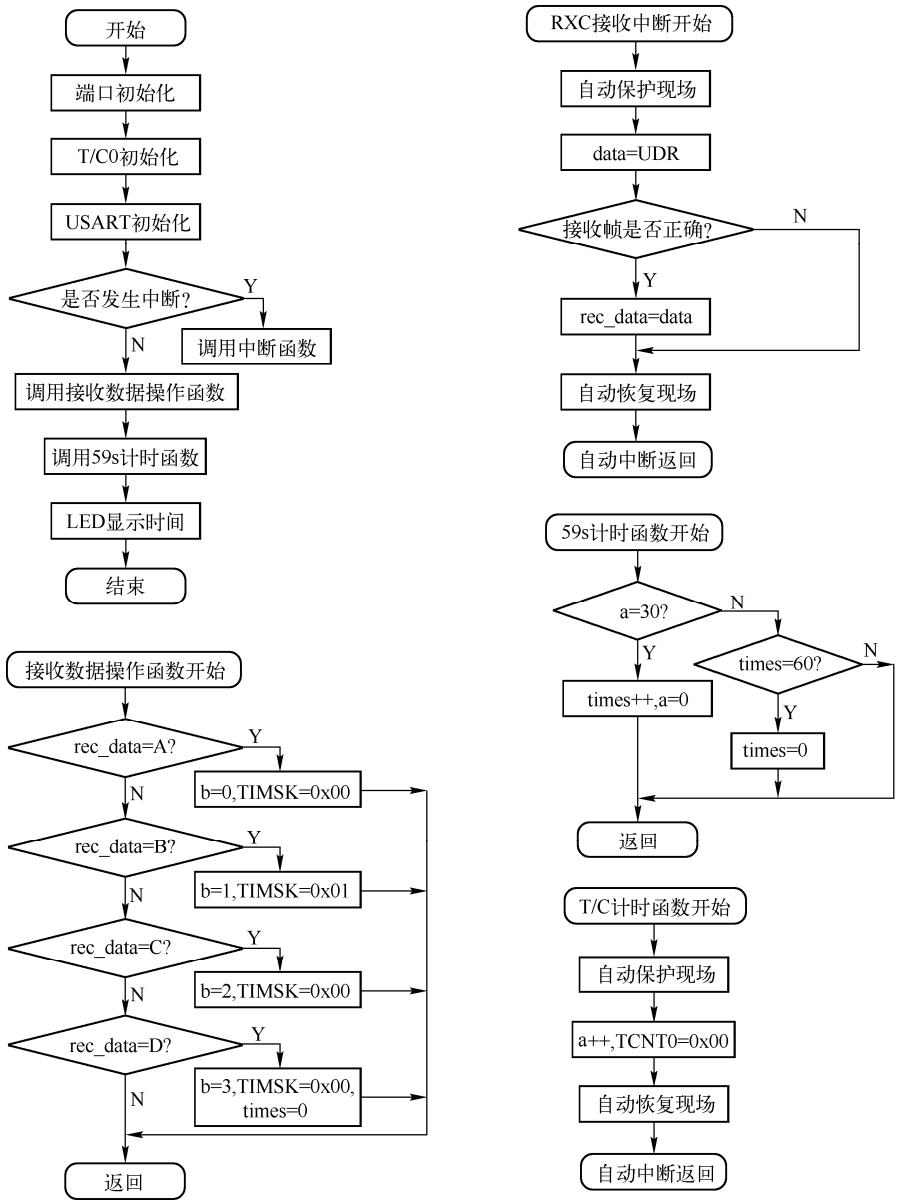
软件设计

本任务也属于单片机双机通信，这两个单片机也都工作在半工状态。由于甲机按键 K1 作为乙机计时的“开始”控制键，按键 K2 作为乙机计时的“暂停”键，按键 K3 作为乙机计时的“复位”键，因此可使用变量 send_data 用来记录按键 K1、K2 和 K3 的状态，然后甲机将与 send_data 内容对应的字符通过 SBUF 发送出去即可完成数据发送操作。假如按键 K1、K2、K3 均未按下，则 send_data 为 0；当按键 K1 按下时，send_data 为 1；当按键 K2 按下时，send_data 为 2；当按键 K3 按下时，send_data 为 3。乙机通过 USARTA 接收字符数据，将接收的字符数据存入 rec_data，再将 rec_data 中的字符转换为与之对应的 b，最后根据 b 值控制计时状态。其程序流程图如图 8-10 所示。乙机程序中，串行口接收数据采用中断方式实现。



（a）U1 程序流程图

图 8-10　甲机通过串口控制乙机计时程序流程图



（b）U2 程序流程图

图 8-10　甲机通过串口控制乙机计时程序流程图（续）

源程序

【U1 发送源程序】

```
/*************************************************
文件名:        U1 发送程序.C
单片机型号:    ATmega16
晶振频率:      8.0MHz
说明:          U1 机将计时开始、暂停、清零命令发送给 U2 机
```

```
**************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define   button1          PIND.2
#define   button2          PIND.3
#define   button3          PINB.2
#define   RXC   7
#define   UDRE 5
uchar   send_data;
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void uart0_send(uchar data)                    //数据发送
{
   while(!(UCSRA&(1<<UDRE)));
   UDR=data;
}
interrupt[EXT_INT0] void int0_isr(void)        //INT0 中断函数
{
  delay(10);
  if(button1==0)                               //再次确认按键，软件防抖
    {
      send_data=1;
    }
}
interrupt[EXT_INT1] void int1_isr(void)        //INT1 中断函数
{
  delay(10);
  if(button2==0)                               //再次确认按钮，软件防抖
    {
      send_data=2;
    }
}
interrupt[EXT_INT2] void int2_isr(void)        //INT2 中断函数
{
  delay(10);
  if(button3==0)                               //再次确认按键，软件防抖
    {
      send_data=3;
    }
}
```

```
void INT_init(void)                          //中断初始化设置
{
    MCUCR=0x0A;                              //INT0、INT1 为下降沿时产生中断请求
  MCUCSR=0x00;                               //INT2 为下降沿时产生中断请求
     GICR=0xE0;                              //INT0、INT1、INT2 允许中断
     #asm("sei")                             //开启总中断
}
void USART_init(void)                        //USART 初始化
{
    UCSRA=0x00;
    UBRRH=0;
    UBRRL=25;                                //系统时钟为 8MHz，波特率为 9600bps
    UCSRB=0x18;
    UCSRC=0x86;
}
void port_init(void)                         //端口初始化
{
     DDRB=0xFB;
    PORTB=0xFF;
     DDRD=0xF3;
    PORTD=0xFF;
}
void main(void)
{
    port_init();
    USART_init();
    INT_init();
    send_data=0;
    while(1)
       {
         switch(send_data)
           {
             case 0:uart0_send('A');break;
             case 1:uart0_send('B');break;
             case 2:uart0_send('C');break;
             case 3:uart0_send('D');break;
           }
       }
}
```

【U2 接收源程序】

```
/*****************************************************
 文件名:          U2 接收程序.C
 单片机型号:      ATmega16
 晶振频率:        8.0MHz
 说明:            U2 机接收开始命令则计时;接收暂停命令则暂停计时;
                      接收清零命令则计时清零;使用 T/C0 计时
```

```
**************************************************/
#include "mega16.h"
#define   uchar unsigned char
#define   uint unsigned   int
#define   CS1          PORTB.4                    //CS1 片选
#define   CS2          PORTB.5                    //CS2 片选
#define   LED          PORTC
#define   UDRE         5
#define   FE           4
#define   PE           2
#define   DOR          3
flash uchar tab[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,  //共阳极 LED0～F 的段码
                   0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E,0xBF,0x10};
uchar rec_data;
uint times;
uchar a,b;
uchar data_L,data_H;
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void   opter(void)                                //根据接收数据进行处理
{
  switch(rec_data)
    {
      case 'A':b=0; TIMSK=0x00; break;            //未启动
      case 'B':b=1; TIMSK=0x01; break;            //启动计时
      case 'C':b=2; TIMSK=0x00; break;            //暂停计时
      case 'D':b=3; TIMSK=0x00; times=0; break;   //计时复位
    }
}
void data_in(void)                                //获取高、低位
{
  data_L=times%10;
  data_H=times/10;
}
void display(void)                                //LED 显示
{
  LED=tab[data_H];
  CS1=1;
  delay(5);
  CS1=0;
  LED=tab[data_L];
```

```
    CS2=1;
    delay(5);
    CS2=0;
}
void    time_cnt(void)                          //59s 计时
{
   if(a==30)
      {
        times++;
        a=0;
        if(times==60)
           {
             times=0;
           }
      }
}
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
   a++;
   TCNT0=0x00;
}
interrupt [USART_RXC] void usart_rx_isr(void)       //USART 串行接收中断
{
    uchar status,data;
    status=UCSRA;
    data=UDR;
    if((status&((1<<FE)|(1<<PE)|(1<<DOR)))==0)
      {
        rec_data=data;
      }
}
void port_init(void)                                //端口初始化
{
    DDRB=0xFF;
    PORTB=0xFF;
    DDRC=0xFF;
    PORTC=0xFF;
}
void USART_init(void)                               //USART 初始化
{
    UCSRA=0x00;
    UBRRH=0;
    UBRRL=25;                                       //系统时钟为 8MHz，波特率为 9600bit/s
    UCSRB=0x90;
    UCSRC=0x86;
    #asm("sei")
}
void TC0_init(void)                                 //T/C0 初始化设置
```

```
{
   TCCR0=0x05;       //WGM01=0、WGM00=0，选择普通模式；COM01=0、COM00=0，使
                     //PB3 为通用 I/O（OC0 不与引脚相连）；CS02=1、CS01=0、CS01=1，
                     //选择 1024 分频
   TCNT0=0x00;
    OCR0=0x00;
}
void main(void)
 {
    port_init();
    USART_init();
    TC0_init();
    while(1)
     {
        opter();
        time_cnt();
        data_in();
        display();
     }
 }
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“U1 发送程序.COFF”和“U2 接收程序.COFF”文件，然后在 Proteus ISIS 中打开已创建的甲机通过串口控制乙机计时电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入程序仿真。按下按键 K1 时，LED 数码管每隔 1s 加 1 显示。当 LED 数码管显示到 59 后，再隔 1s 时，将显示为 0，并继续计时加 1 显示。按下按键 K2 时，暂停计时加 1，LED 数码管将显示暂停前的数值。再次按下按键 K1 时，继续计时加 1 显示。按下按键 K3 时，LED 数码管将立即显示为 0，并停止计时。



任务 4　单片机双机通信

单片机 U1（甲机）的 PD2 外接按键 K1，PC、PB4、PB5 端口作为输出口，外接一个 2 位 LED 数码管；单片机 U2（乙机）PD 外接按键 K2，PC 端口外接 8 位 LED。要求使用单片机串行通信，甲机按键 K1 每按下 1 次，乙机的 LED 移位 1 次；乙机按键 K2 每按下 1 次，甲机的 LED 数码管加 1 显示。



硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 8-12 所列的元器件，并完成图 8-11 所示的硬件电路图设计。

表 8-12　单片机双机通信所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	数码管 7SEG-MPX2-CA	电解电容 CAP-ELEC
电阻 RES	按钮 BUTTON	发光二极管 LED-RED	发光二极管 LED-YELLOW
电平转换 MAX232	DB-9　CONN-D9F	发光二极管 LED-BLUE	发光二极管 LED-GREEN
三极管 NPN	晶振 CRYSTAL 8MHz		

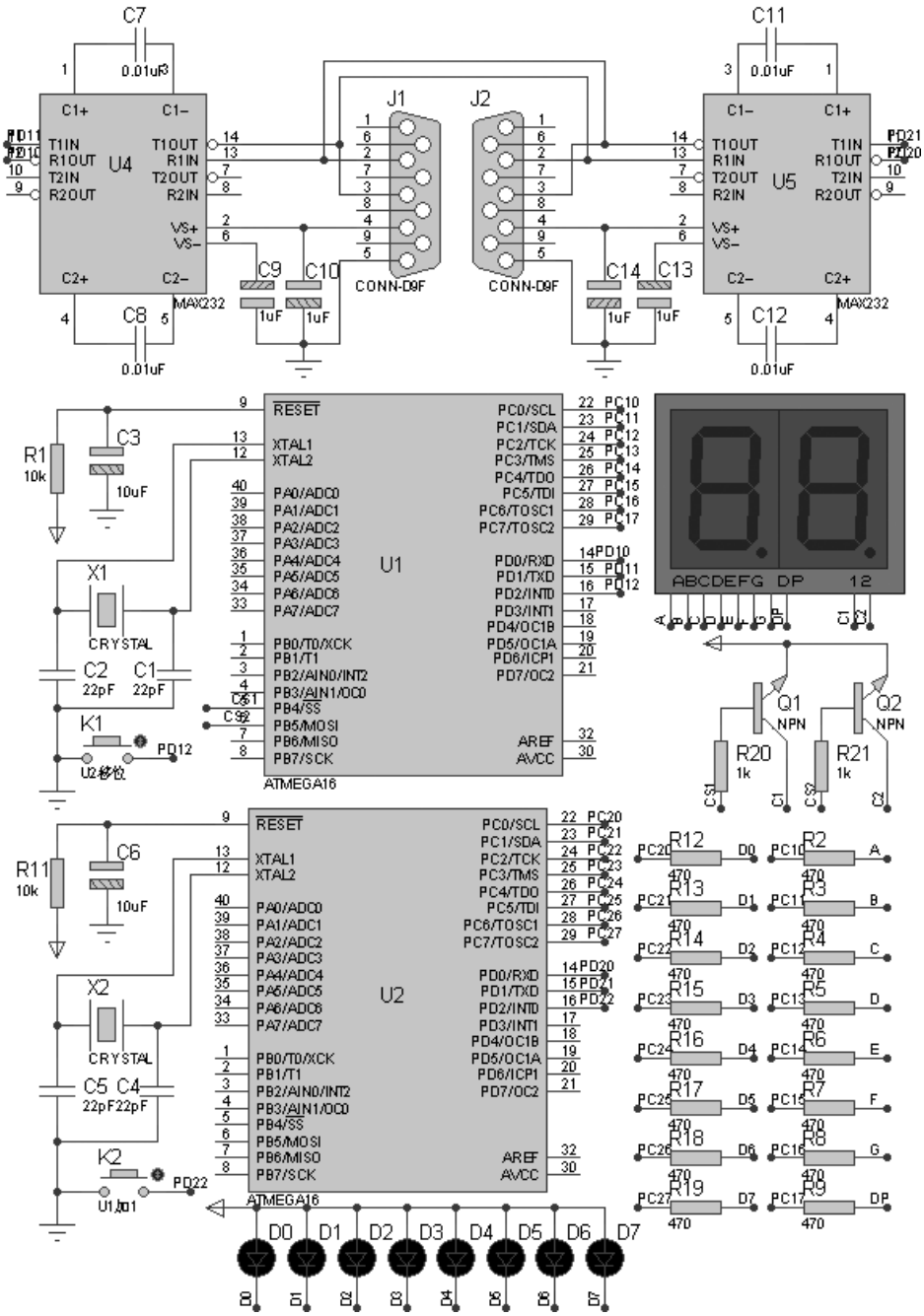


图 8-11　单片机双机通信电路图

软件设计

本任务属于全双工单片机双机通信，甲机和乙机都能发送和接收数据。

【甲机源程序设计分析】甲机源程序主要包括数据发送、数据接收、INT0 中断和主程序等部分。这 4 个部分相应的函数参考前面的相关知识即可，在此主要讲述主程序的设计方法。

主程序中的初始化包括对 PB、PC、PD 端口的初始化，INT0 相关寄存器的初始化和 USART 相关寄存器的初始化。

> 注意:
> 对 USART 相关寄存器进行初始化时，一定要设置 USART 为既能发送又能接收数据帧，且甲机与乙机的传输速率应完全一致，在此将波特率设置为 9600bit/s。初始化后要开启全局中断，以确 INT0 和 USART 均能响应中断。

在主程序中除了进行相应的初始化外，还要进行 3 个方面的操作：获取按键 K1 状态、根据接收的数据决定相应任务、数据显示。首先获取按键 K1 的状态，并将这些状态值送到发送函数中，然后通过 uart0_send()函数将这些值由 TXD 发送到乙机，以控制乙机 LED 的显示状态。甲机通过 RXD 接收乙机发送过来的 rec_data 数据，这些数据决定数码管显示内容。其程序流程图如图 8-12（a）所示。

【乙机源程序设计分析】乙机源程序主要包括数据发送、数据接收、外部中断 INT0 和主程序这 4 部分。

在外部中断 INT0，主要是获取 a 值，这些值先送到 USART 的 UDR 中，再通过 TXD 发送给甲机，以控制甲机定时器的开启/暂停和复位。

主程序中的初始化包括 PC、PD 端口的初始化，INT0 的相关寄存器的初始化和 USART 相关寄存器的初始化。

> 注意:
> 对 USART 相关寄存器进行初始化时，一定要设置 USART 为既能发送又能接收数据帧，且乙机与甲机的传输速率应完全一致，在此将波特率设置为 9600bit/s。初始化后要开启全局中断，以确保外部中断 INT0、INT1 和 USART 均能响应中断。

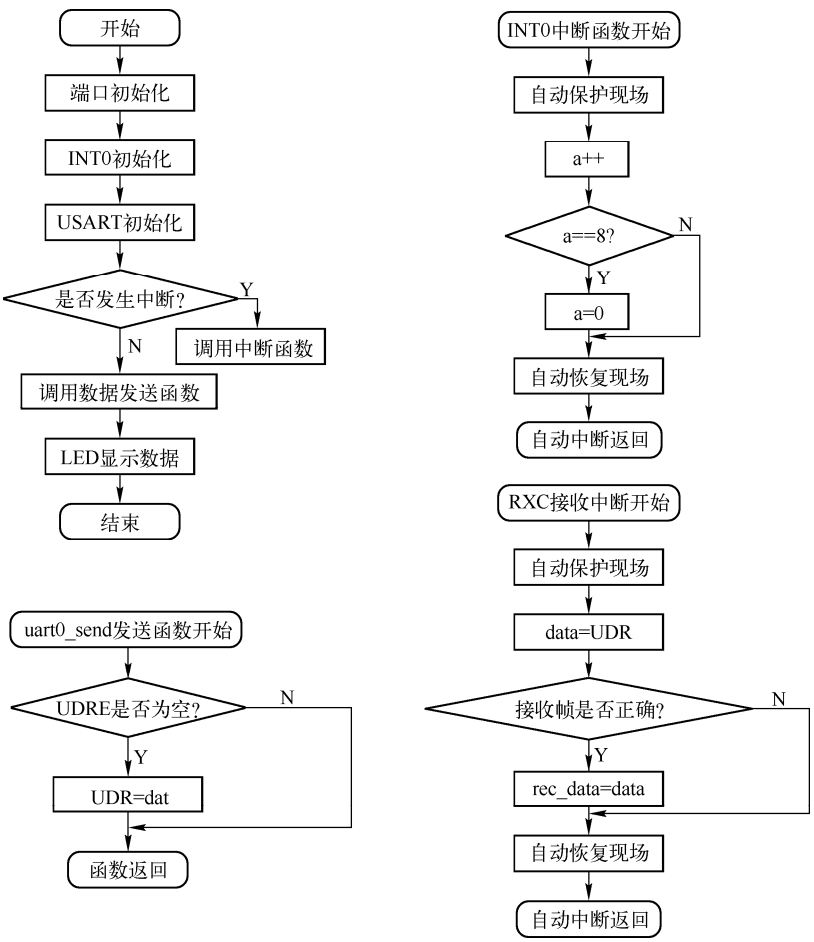
在主程序中除了进行相应的初始化外，还要进行两个方面的操作。首先将 INT0 产生的 a 值通过发送函数发送给甲机，然后将接收过来的数据 rec_data 送到 PC 端口即可。其程序流程图如图 8-12（b）所示。

源程序

【U1 源程序】

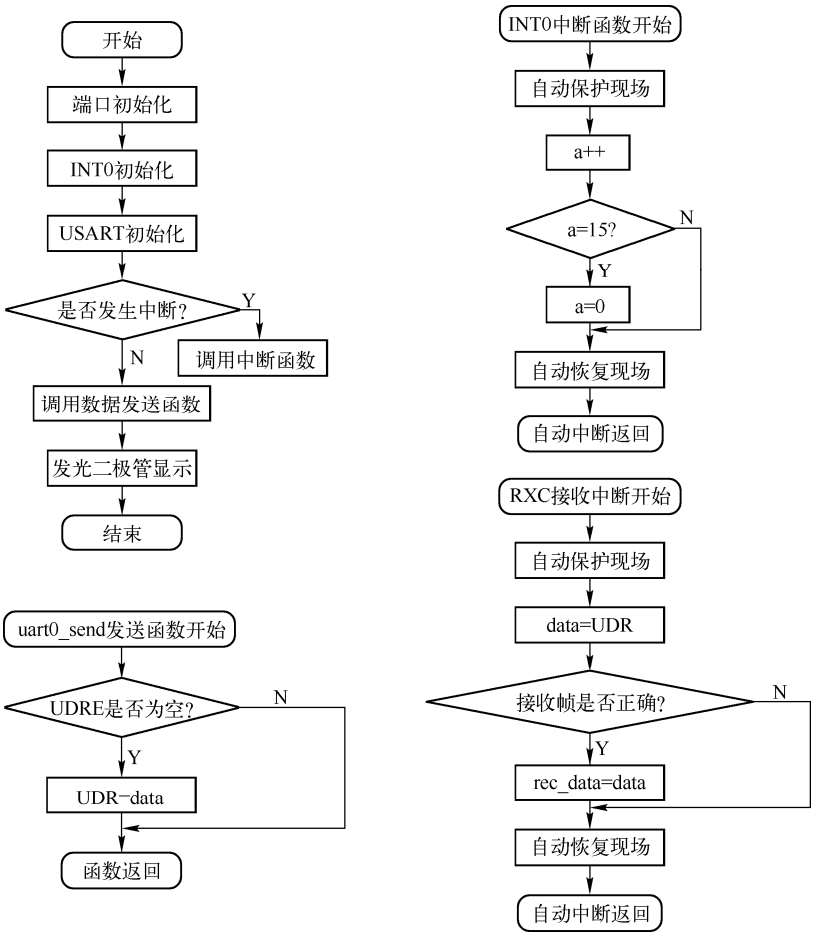
```
/****************************************************
文件名:          U1 程序.C
单片机型号:      ATmega16
晶振频率:        8.0MHz
说明:            U1 控制 U2 的 LED 流水灯，接收 U2 数码管显示数据
```

```
**************************************************/
#include "mega16.h"
#define   uchar unsigned char
#define   uint unsigned   int
#define   CS1              PORTB.4              //CS1 片选
#define   CS2              PORTB.5              //CS2 片选
#define   button           PIND.2               //CS2 片选
#define   LED              PORTC
#define   UDRE             5
#define   FE               4
#define   PE               2
#define   DOR              3
flash uchar tab[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,   //共阳极 LED0～F 的段码
                   0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E,0xBF,0x10};
uchar rec_data;
uchar a;
uchar data_L,data_H;
void delay(uint k)
```



（a）U1 程序流程图

图 8-12　单片机双机通信程序流程图



（b）U2 程序流程图

图 8-12　单片机双机通信程序流程图（续）

```
{
   uint m,n;
   for(m=0;m<k;m++)
      {
         for(n=0;n<1140;n++)
           {;}
      }
}
void data_in(void)                              //获取高、低位
{
   data_L=rec_data%10;
   data_H=rec_data/10;
}
void display(void)                              //LED 显示
{
   LED=tab[data_H];
   CS1=1;
```

```
    delay(5);
    CS1=0;
    LED=tab[data_L];
    CS2=1;
    delay(5);
    CS2=0;
}
interrupt[EXT_INT0] void int0_isr(void)          //INT0 中断函数
{
    delay(10);
    if(button==0)                                //再次确认按键，软件防抖
      {
          a++;
          if(a==8)
            {
               a=0;
            }
      }
}
interrupt [USART_RXC] void usart_rx_isr(void)    //USART 串行接收中断
{
    uchar status,data;
    status=UCSRA;
    data=UDR;
    if((status&((1<<FE)|(1<<PE)|(1<<DOR)))==0)
     {
        rec_data=data;
     }
}
void uart0_send(uchar dat)                       //数据发送
{
    while(!(UCSRA&(1<<UDRE)));
    UDR=dat;
}
void port_init(void)                             //端口初始化
{
    DDRB=0xFF;
    PORTB=0xFF;
    DDRC=0xFF;
    PORTC=0xFF;
    DDRD=0xFB;
    PORTD=0xFC;
}
void INT_init(void)                              //中断初始化设置
{
    MCUCR=0x02;                                  //INT0 为下降沿时产生中断请求
     GICR=0x40;                                  //INT0 允许中断
     #asm("sei")                                 //开启总中断
```

```
}
void USART_init(void)                          //USART 初始化
{
    UBRRH=0;
    UBRRL=25;                                  //系统时钟为 8MHz，波特率为 9600bit/s
    UCSRA=0x00;
    UCSRB=0x98;
    UCSRC=0x86;
}
void main(void)
 {
    port_init();
    INT_init();
    USART_init();
    while(1)
     {
        uart0_send(a);
        data_in();
        display();
     }
 }
```

【U2 源程序】

```
/****************************************************
 文件名:            U2 程序.C
 单片机型号:        ATmega16
 晶振频率:          8.0MHz
 说明:              U2 控制 U1 的数码管显示数据,接收 U1 的 LED 移位数据
****************************************************/
#include "mega16.h"
#define   uchar unsigned char
#define   uint unsigned   int
#define   button          PIND.2
#define   LED             PORTC
#define   UDRE            5
#define   FE              4
#define   PE              2
#define   DOR             3
flash uchar discode[]={0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80};
uchar rec_data;
uchar a;
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
       for(n=0;n<1140;n++)
```

```
            {;}
        }
}
void display(void)                          //LED 显示
{
   LED=~discode[rec_data];
}
interrupt[EXT_INT0] void int0_isr(void)     //INT0 中断函数
{
   delay(10);
   if(button==0)                            //再次确认按键，软件防抖
      {
          a++;
          if(a==15)
            {
               a=0;
            }
      }
}
interrupt [USART_RXC] void usart_rx_isr(void)   //USART 串行接收中断
{
    uchar status,data;
    status=UCSRA;
    data=UDR;
    if((status&((1<<FE)|(1<<PE)|(1<<DOR)))==0)
     {
        rec_data=data;
     }
}
void uart0_send(uchar dat)                  //数据发送
{
    while(!(UCSRA&(1<<UDRE)));
    UDR=dat;
}
void port_init(void)                        //端口初始化
{
    DDRC=0xFF;
    PORTC=0xFF;
    DDRD=0xFB;
    PORTD=0xFC;
}
void INT_init(void)                         //中断初始化设置
{
    MCUCR=0x02;                             //INT0 为下降沿时产生中断请求
     GICR=0x40;                             //INT0 允许中断
     #asm("sei")                            //开启总中断
}
void USART_init(void)                       //USART 初始化
```

```
{
    UBRRH=0;
    UBRRL=25;                                    //系统时钟为 8MHz，波特率为 9600bit/s
    UCSRA=0x00;
    UCSRB=0x98;
    UCSRC=0x86;
}
void main(void)
 {
    port_init();
    INT_init();
    USART_init();
    while(1)
     {
        uart0_send(a);
        display();
     }
 }
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“U1 程序.COFF”和“U2 程序.COFF”文件，然后在 Proteus ISIS 中打开已创建的单片机双机通信电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入程序仿真。每按一次按键 K1 时，LED 数码管加 1 显示。当 LED 数码管显示到 15 后，再次按下按键 K1 时，将显示为 0。每按一次按键 K2 时，LED 进行移位 1 次显示。



任务 5　单片机自发自收数据

在项目六的任务 7 中讲述了利用 INT0～INT2 这 3 个中断实现计数控制，现要求在此基础上，计数值先通过单片机的 TXD 端串行输出，再通过 RXD 端输入该数据，然后驱动两位 LED 数码管进行数据显示。



硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 8-13 所列的元器件，并完成如图 8-13 中所示的硬件电路图设计。

表 8-13　单片机自发自收数据所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	电解电容 CAP-ELEC	按钮 BUTTON
电阻 RES	晶振 CRYSTAL 8MHz	数码管 7SEG-MPX2-CA-BULE	三极管 NPN

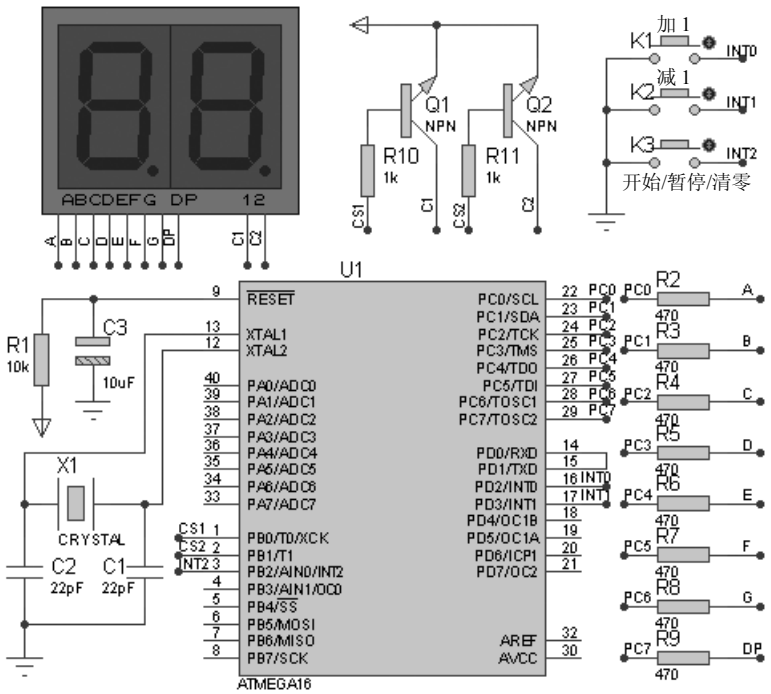


图 8-13　单片机自发自收数据电路图

软件设计

在使用 USART 时，首先要根据实际使用的要求和规定，对它进行正确的初始化设置。数据的发送和接收的实现，可采用查询或中断的方式进行。本设计的发送采用查询方式，接收采用中断方式。如果数据传输速率为 9600bit/s，由于系统时钟为 8MHz，所以 USART 串行数据传输的初始化设置如下：UCSRA=0x00；UCSRB=0x98，即 RXCIE=1，允许使用接收中断；REXN=1，启动 USART 接收器；TXEN=1，启动 USART 发送器；UCSZ2=0；UCSRC=0x86，即 URSEL=1，使用 UCSRC；UMSEL=0，选择异步模式；UPM1=0、UPM0=0，禁止奇偶校验；UCSZ1=1、UCSZ0=1，选择字符长度为 8 位；UBRRH=0x00，UBRRL=25，波特率为 9600bit/s。

该设计采用 4 个中断函数：INT0～INT2 外部中断和 RXC 接收中断函数，其中 INT0～INT2 的中断函数的编写与项目六的任务 7 中编写的完全相同。在 RXC 接收中断函数中，先将接收的数据 UDR 送入 data，再通过 UCSRA 判断在传输中是否发生帧错误、有数据溢出或发生奇偶校验错误，如果没发生这些错误时再将 data 送入 count。LED 七段数码管根据 count 中的值进行相应的显示即可。

数据发送时，可编写一个发送查询函数。在此函数中，主要判断 USART 的 UDRE 标志是否为空，若为空，则将 dat 送入 UDR。采用这种方法，先在主程序中调用该函数，然后将 a 送给 dat 即可完成多个数据的传送。其程序流程图如图 8-14 所示，而 INT0～INT2 中的程序流程见项目六的任务 7。

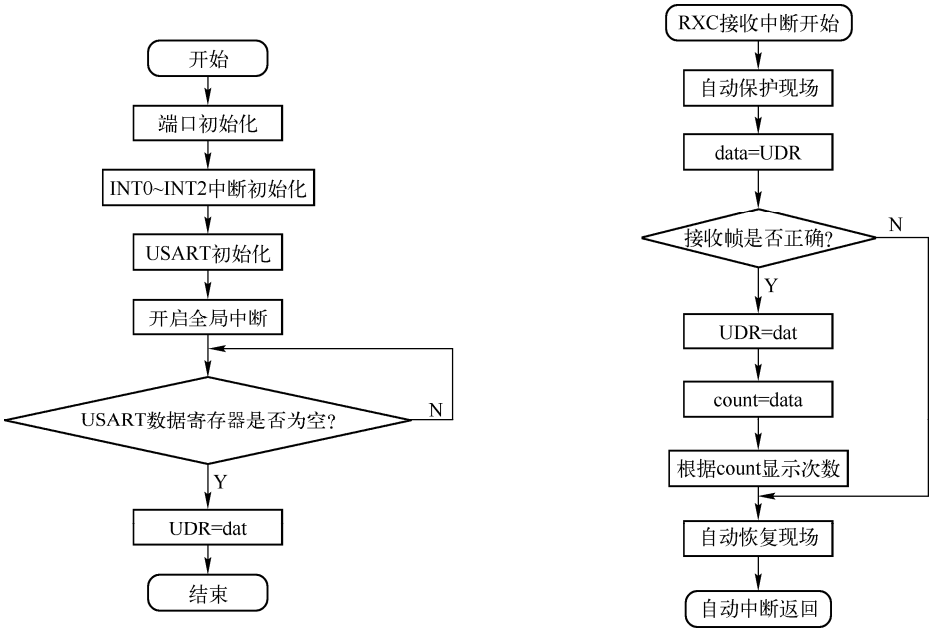


图 8-14　单片机自发自收数据程序流程图

源程序

```
/*****************************************************
  文件名:          单片机自发自收数据.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
  说明:            使用串行通信和外部中断实现计数和清零控制
*****************************************************/
#include "mega16.h"
#define   uchar unsigned char
#define   uint unsigned int
#define   UDRE          5
#define   FE            4
#define   PE            2
#define   DOR           3
#define   button1       PIND.2
#define   button2       PIND.3
#define   button3       PINB.2
#define   CS1           PORTB.0                    //CS1 片选
#define   CS2           PORTB.1                    //CS2 片选
#define   LED           PORTC
flash uchar tab[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,    //共阳极 LED0～F 的段码
                   0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E,0xBF,0x10};
uchar a;                                             //外部中断计数值
uchar count;                                         //串行通信接收中断计数值
uchar data_H,data_L;                                 //计数值高、低位
```

```
uchar b;                                          //开始/暂停/清零选择
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void data_in(void)                                //获取高、低位
{
  data_L=count%10;
  data_H=count/10;
}
void display(void)
{
  LED=tab[data_H];
  CS1=1;
  delay(5);
  CS1=0;
  LED=tab[data_L];
  CS2=1;
  delay(5);
  CS2=0;
}
interrupt[EXT_INT0] void int0_isr(void)           //INT0 中断函数
{
  delay(10);
  if(button1==0)                                  //再次确认按键，软件防抖
    {
      if(a==99)
        {
          a=0;
        }
      else
        {
          a++;
        }
    }
}
interrupt[EXT_INT1] void int1_isr(void)           //INT1 中断函数
{
  delay(10);
  if(button2==0)                                  //再次确认按键，软件防抖
    {
      if(a==0)
```

```
            {
              a=99;
            }
          else
            {
              a--;
            }
        }
}
interrupt[EXT_INT2] void int2_isr(void)                    //INT2 中断函数
{
  delay(10);
  if(button3==0)                                           //再次确认按键，软件防抖
    {
      b++;
      if(b==1)
        {                                                  //允许计数
          GICR=0xE0;                                       //允许 INT0、INT1、INT2 产生中断
        }
      if(b==2)                                             //暂停计数
        {
          GICR=0x20;                                       //只允许 INT2 产生中断
          delay(100);
          GICR=0xE0;
        }
      if(b>=3)                                             //清零
        {
          b=0;
          a=0;
          GICR=0x20;                                       //只允许 INT2 产生中断
        }
    }
}
interrupt [USART_RXC] void usart_rx_isr(void)              //USART 串行接收中断
 {
   uchar status,data;
   status=UCSRA;
   data=UDR;
   if((status&((1<<FE)|(1<<PE)|(1<<DOR)))==0)
    {
      count=data;
    }
 }
void uart0_send (char dat)                                 //USART 串行发送数据
{
   while( !(UCSRA & (1 << UDRE)) );                        //只有数据寄存器为空时才能发送数据
UDR = dat;
```

```
}

void port_init(void)                    //端口初始化
{
     DDRB=0xFB;                         //PB 端口配置为输出
   PORTB=0xFF;                          //PB 端口初始值设置
     DDRC=0xFF;                         //PC 端口配置为输出
   PORTC=0xFF;                          //PC 端口初始值设置
     DDRD=0xF2;                         //PD 端口配置为输出
   PORTD=0xFF;                          //PD 端口初始值设置
}
void INT_init(void)                     //中断初始化设置
{
    MCUCR=0x00;                         //INT0、INT1 为低电平时产生中断请求
  MCUCSR=0x00;                          //INT2 为低电平时产生中断请求
     GICR=0x20;                         //允许 INT2 产生中断
     #asm("sei")                        //开启总中断
}
void USART_init(void)                   //USART 初始化
{
    UBRRH=0;
    UBRRL=25;                           //系统时钟为 8MHz，波特率为 9600bit/s
    UCSRA=0x00;
    UCSRB=0x98;
    UCSRC=0x86;
}
void main(void)
{
  port_init();
  INT_init();
  USART_init();
  a=0;b=0;
  while(1)
     {
       uart0_send (a);
       data_in();
       display();
     }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“单片机自发自收数据.COFF”文件，然后在 Proteus ISIS 中打开已创建的单片机自发自收数据电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入程序仿真。在按键 K1、K2、K3 都没有按下的初始状态时，LED 数码管显示 00。如果按键 K3 没有按下，不管按键 K1 或 K2 按下多少次，LED 数码管仍然显示为 00。第 1 次按下按键 K2 后，每次按下按键 K1，LED 数码

管进行加 1 显示，当显示达 99 时，按键 K1 再次按下，LED 数码管显示为 0；每次按下按键 K2，LED 数码管进行减 1 显示，当显示达 00 时，按键 K2 再次按下，LED 数码管显示为 99。第 2 次按下按键 K2 时，暂停加 1/减 1 计数，按键 K1 或 K2 按下后 LED 数码管显示暂停前的数值，当延时 0.1s 后，恢复加 1/减 1 计数。第 3 次按下按键 K2 时，LED 数码管显示为 00，并停止加 1/减 1 计数。第 4 次按下按键 K2 时，每次按下按键 K1，LED 数码管进行加 1 显示。

项目九　LED 点阵显示器的设计

〖知识目标〗

☺ 掌握 LED 点阵显示器的工作原理和字符及汉字的显示方法;

☺ 学会元器件 74LS138、74HC154 的原理和使用方法。

〖能力目标〗

☺ 能够编写点阵式 LED 的字符、汉字显示程序;

☺ 会用 CVAVR 软件对源程序进行编译调试及与 Proteus 软件联调，实现电路仿真。

任务 1　LED 点阵显示器控制原理认知

1．LED 点阵显示器构成及显示原理

LED 点阵显示器由一串发光或不发光的点状（或条状）显示器按矩阵的方式排列组成的，其发光体是 LED。现在点阵显示器应用十分广泛，如广告活动字幕机、股票显示屏、活动布告栏等。

LED 点阵显示器的分类有多种方法：按阵列点数可分为 5×7、5×8、6×8、8×8 四种；按发光颜色可分为单色、双色、三色；按极性排列方式又可分为共阳极和共阴极。图 9-1 所示为 5×7 的共阴极和共阳极阵列结构。

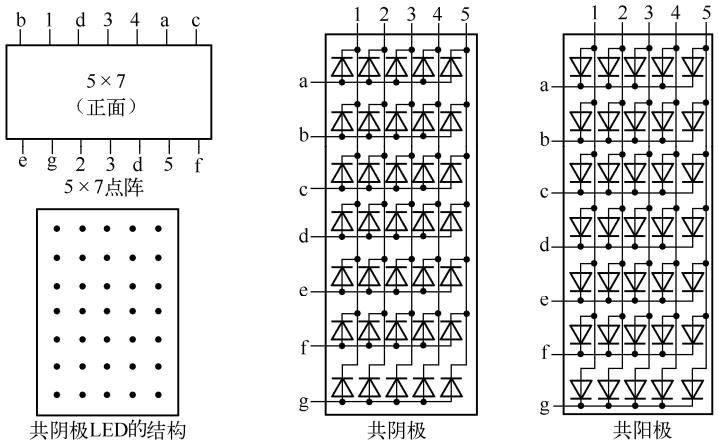


图 9-1　共阴极和共阳极阵列结构

从图 9-1 可以看出，只要让某些 LED 点亮，就可组成数字、字母、图形、汉字等。显示单个字母、数字时，只需一个 5×7 的 LED 点阵显示器即可，如图 9-2 所示。显示汉字时，需多个 LED 点阵显示器组合，最常见的组合方式有 15×14、16×15、16×16 等。

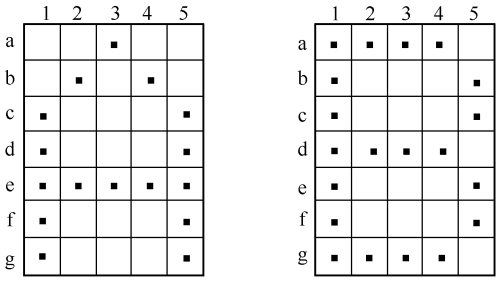


图 9-2　LED 点阵显示字母“A”和“B”

2．LED 点阵显示器字模软件的使用

为了在 LED 点阵上显示不同的字符，用户在编程时需要编写相关的字符代码。通常这些字符代码可以由用户通过点阵方式手工生成，也可以通过字模软件生成。当前，有许多的字模软件可供用户使用，如 PCtoLCD2002、字模 III-增强版等。

PCtoLCD2002 是一种字模生成软件，它同样适用于为 LED 点阵显示器生成字模。该软件不需安装，在程序包中直接双击 PCtoLCD2002 文件，即可启动该软件，如图 9-3 所示。



图 9-3　启动 PCtoLCD2002 软件

生成字模的主要步骤如下所述。

1）选择模式　在 PCtoLCD2002 软件中执行菜单命令“模式”→“字符模式”。

2）字模选择设置　在 PCtoLCD2002 软件中单击“选项”菜单栏，将弹出如图 9-4 所示的“字模选项”对话框。在此对话框中，可根据实际情况主要设置点阵格式、取模方式、取模走向、自定义格式。

3）选择字体、字宽和字高　在“请选择字体”栏的下拉列菜单中可以选择合适的字体。在字宽和字高中分别可以设置字符的宽度和高度。



图 9-4 “字模选项”对话框

4）输入字符 在“生成字模”按钮的左侧空白栏中，输入需要转换的字符。

5）生成字模 输入字符后，单击“生成字模”按钮，在该按钮的下方空白窗口中将会显示相应的字符代码，用户直接可以将代码进行复制即可。

任务 2 一个 5×7 LED 点阵字符显示

使用一个 5×7 共阴极 LED 点阵显示字符“E”。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 9-1 所列的元器件，并完成图 9-5 所示的硬件电路图设计。

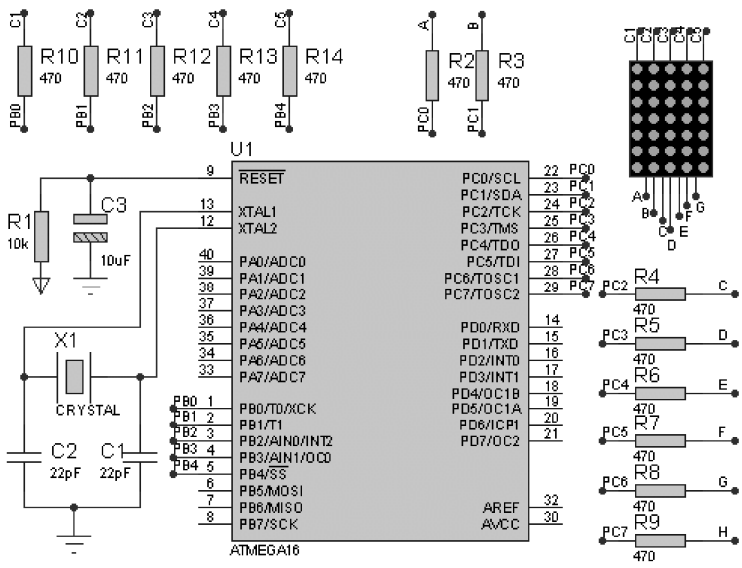


图 9-5 一个 5×7 点阵字符显示电路图

表 9-1　一个 5×7 点阵字符显示所使用的元器件

单片机 ATmega16	电解电容 CAP-ELEC 10uF	瓷片电容 CAP22pF	电阻 RES
晶振 CRYSTAL 8MHz	点阵 MATRIX-5X7-GREEN		

软件设计

5×7 共阴极 LED 点阵的段码 a～g 是高电平有效，位选高电平有效。因此显示字符“E”的段码值如图 9-6 所示。

5×7 共阴极 LED 点阵显示字符“B”，可以通过建立一个数据表格的形式进行，首先位选 1 有效，将段码值 0x00 送给 PORTC 端口以驱动相应段点亮，然后位选 2 有效，将段码值 0x36 送给 PORTC 端口以驱动相应段点亮……如此进行，直到送完最后一列段码，又从头开始，因此其程序流程图如图 9-7 所示。

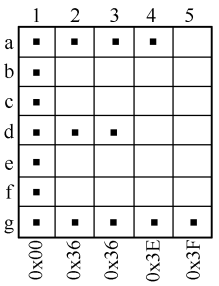


图 9-6 “E”字模

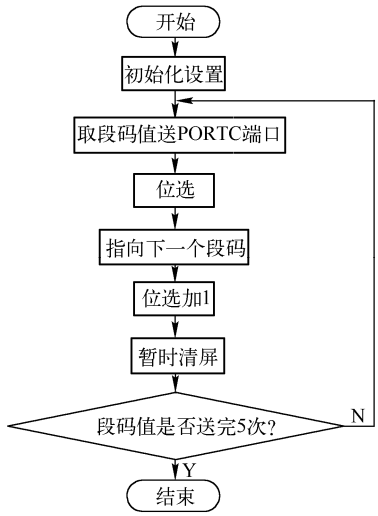


图 9-7　5×7 共阴极 LED 点阵显示流程图

源程序

```
/************************************************************
  文件名:          一个 5×7LED 点阵字符显示.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
  说明:            在 5×7LED 点阵上显示字符 E
************************************************************/
#include "mega16.h"
#define   uchar unsigned char
#define   uint unsigned   int
#define   CS1             PORTB.0              //CS1 片选
#define   CS2             PORTB.1              //CS2 片选
#define   CS3             PORTB.2              //CS3 片选
#define   CS4             PORTB.3              //CS4 片选
#define   CS5             PORTB.4              //CS5 片选
```

```
#define   CS                  PORTB
#define   LED                 PORTC
flash uchar tab1[]={0x00,0x36,0x36,0x3E,0x3F};          //E 字库
flash uchar tab2[]={0x01,0x02,0x04,0x08,0x10};          //扫描代码
uchar i,j;
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void display(void)
{
   for(i=0;i<5;i++)
     {
       CS=tab2[j];
      LED=tab1[i];
      delay(2);
      j++;
      if(j= =5)
        {
          j=0;
          CS=0x00;
        }
     }
}
void port_init(void)                                    //端口初始化
{
    DDRB=0xFF;
   PORTB=0xFF;
    DDRC=0xFF;
   PORTC=0xFF;
}
void main(void)
{
   port_init();
   while(1)
     {
       display();
     }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“一个 5×7 点阵字符显示.COFF”文件，然后在 Proteus ISIS 中打开已创建的一个 5×7 点阵字符显示电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试运行状态下，可

看见在 5×7 点阵显示字符“E”，运行效果如图 9-8 所示。

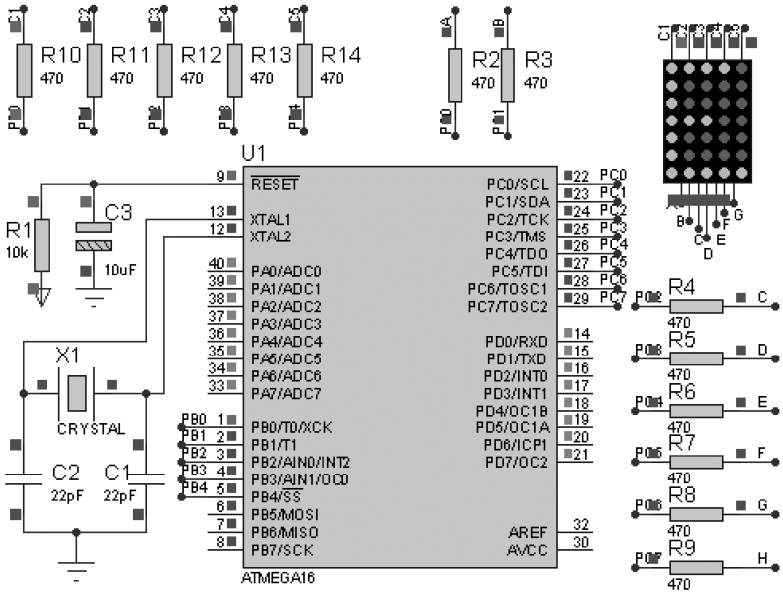


图 9-8　一个 5×7 点阵字符显示仿真效果图

任务 3　一个 8×8 LED 点阵字符串显示

1．一个 8×8 LED 点阵字符串显示（一）

采用列行式的取模方式，在一个 8×8 LED 点阵中显示字符串“I like the ATmega16 MCU”。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 9-2 所列的元器件，并完成图 9-9 所示的硬件电路图设计。

表 9-2　一个 8×8 点阵字符串显示（一）所使用的元器件

单片机 ATmega16	电解电容 CAP-ELEC 10uF	瓷片电容 CAP22pF	电阻 RES
晶振 CRYSTAL 8MHz	点阵 MATRIX-8X8-GREEN	3-8 线译码器 74LS138	

软件设计

一个 8×8 LED 点阵在某一时刻只能显示一个字符，要显示字符串，必须在显示完一个字符后接着显示下一个字符。由于每个字符有 8 个段码值，该字符串有 23 个字符，所以该字符串库中有 8×23 共 184 个段码值。使用 PCtoLCD2002 字模软件，选择点阵格式为共阴极、取模方式为列行式、取模方向为高位在前，可生成这 15 个字符的字模段码值。

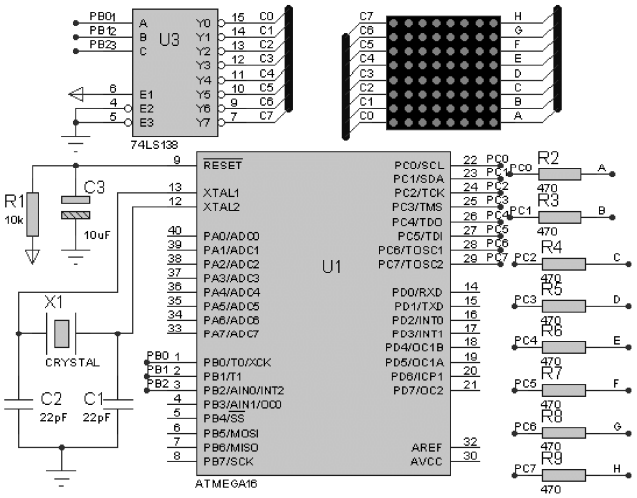


图 9-9　一个 8×8 LED 点阵字符串显示（一）电路图

8×8 共阴极 LED 点阵显示字符串“I like the ATmega16 MCU”，可以通过建立两个一维数组的形式进行，单片机的 P0 端口输出段码值，PORTB 端口输出位选值，并通过 74LS138 译码从而控制单片机对 LED 点阵进行位选。首先位选 1 有效，将段码值 0x00 送给 PORTC 端口以驱动相应段点亮，然后位选 2 有效，将段码值 0x22 送给 PORTC 端口以驱动相应段点亮……如此进行，直到送完 8 个段码，就可显示“I”，然后再进行字符 1 的显示……每个字符的显示与字符“I”的显示过程相同，只是段码值不同而已。每送一个段码值时均有相应的计数值，当计数值达到 184 时，表示“I like the ATmega16 MCU”字符串已显示完，又从头开始，因此该程序流程图如图 9-10 所示。

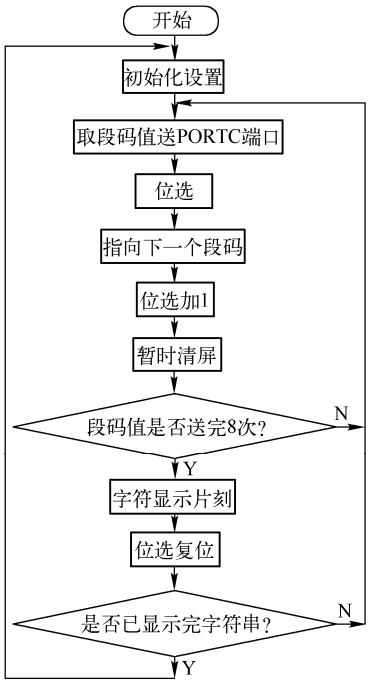


图 9-10　一个 8×8 点阵字符串显示（一）程序流程图

源程序

```
/************************************************************
 文件名:          一个 8×8LED 点阵字符串显示（一）.c
 单片机型号:      ATmega16
 晶振频率:        8.0MHz
 说明:            在 8×8LED 点阵上显示字符串 I like the ATmega16 MCU
 取模方法:        宋体、共阴极、取模式方式为列行式、高位在前
************************************************************/
#include "mega16.h"
#define   uchar unsigned char
#define   uint unsigned   int
#define   CS1              PORTB.0                    //CS1 片选
#define   CS2              PORTB.1                    //CS2 片选
#define   CS3              PORTB.2                    //CS3 片选
#define   CS               PORTB
#define   LED               PORTC
uint   j,i,q=0,t=0;
flash uchar tab1[]={
      0x00,0x22,0x22,0x3E,0x22,0x22,0x00,0x00,          /*"I",0*/
      0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,          /*" ",1*/
      0x00,0x22,0x22,0x7E,0x02,0x02,0x00,0x00,          /*"l",2*/
      0x00,0x12,0x32,0x3E,0x02,0x02,0x00,0x00,          /*"i",3*/
      0x22,0x3E,0x06,0x0C,0x1E,0x12,0x12,0x00,          /*"k",4*/
      0x00,0x0C,0x1A,0x1A,0x1A,0x1A,0x0A,0x00,          /*"e",5*/
      0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,          /*" ",6*/
      0x00,0x10,0x10,0x3E,0x12,0x12,0x00,0x00,          /*"t",7*/
      0x22,0x3E,0x12,0x10,0x10,0x12,0x1E,0x02,          /*"h",8*/
      0x00,0x0C,0x1A,0x1A,0x1A,0x1A,0x0A,0x00,          /*"e",9*/
      0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,          /*" ",10*/
      0x02,0x06,0x1E,0x34,0x14,0x0E,0x02,0x02,          /*"A",11*/
      0x20,0x20,0x22,0x3E,0x22,0x20,0x20,0x00,          /*"T",12*/
      0x12,0x1E,0x12,0x10,0x1E,0x12,0x10,0x1E,          /*"m",13*/
      0x00,0x0C,0x1A,0x1A,0x1A,0x1A,0x0A,0x00,          /*"e",14*/
      0x00,0x1F,0x1D,0x1B,0x1B,0x1B,0x13,0x00,          /*"g",15*/
      0x00,0x16,0x1A,0x1A,0x1A,0x1A,0x1E,0x02,          /*"a",16*/
      0x00,0x12,0x12,0x3E,0x02,0x02,0x00,0x00,          /*"1",17*/
      0x00,0x1C,0x2A,0x2A,0x2A,0x2A,0x04,0x00,          /*"6",18*/
      0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,          /*" ",19*/
      0x22,0x3E,0x38,0x0E,0x30,0x3E,0x22,0x00,          /*"M",20*/
      0x1C,0x36,0x22,0x22,0x22,0x22,0x24,0x00,          /*"C",21*/
      0x20,0x3E,0x22,0x02,0x02,0x22,0x3E,0x20};         /*"U",22*/
flash uchar tab2[]={0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07};    //扫描代码
void delay(uint k)
{
   uint m,n;
   for(m=0;m<k;m++)
      {
```

```
            for(n=0;n<1140;n++)
              {;}
          }
    }
    void display(void)
    {
       for(i=0;i<15;i++)
           {
              for(j=q;j<8+q;j++)
                 {
                    CS=tab2[t++];
                  LED=tab1[j];
                  delay(4);
                  if(t= =8)
                     {
                        t=0;
                     }
                 }
           }
     q=q+8;
     if(q= =184)
            {
              q=0;
            }
    }
    void port_init(void)                                    //端口初始化
    {
          DDRB=0xFF;
        PORTB=0xFF;
          DDRC=0xFF;
        PORTC=0xFF;
    }
    void main(void)
    {
        port_init();
        while(1)
           {
              display();
           }
    }
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“一个 8×8 点阵字符串显示（一）.COFF”文件，然后在 Proteus ISIS 中打开已创建的一个 8×8 点阵字符串显示（一）电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试运行状态下，可看见 8×8 点阵显示字符串“I like the ATmega16 MCU”，运行效果如图 9-11 所示。

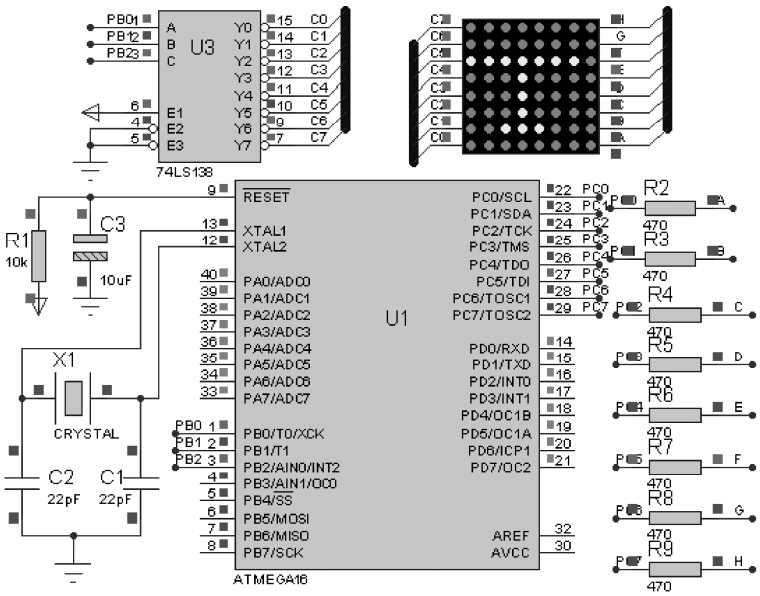


图 9-11　一个 8×8 点阵字符串显示（一）仿真效果图

2．一个 8×8 LED 点阵字符串显示（二）

采用行列式的取模方式，在一个 8×8 LED 点阵中显示字符串“www.phei.com.cn”。



硬件设计

由于取模方式与一个 8×8 LED 点阵字符串显示（一）的取模方式，所以其硬件电路也略有不同。在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 9-2 所列的元器件，并完成图 9-12 所示的硬件电路图设计。

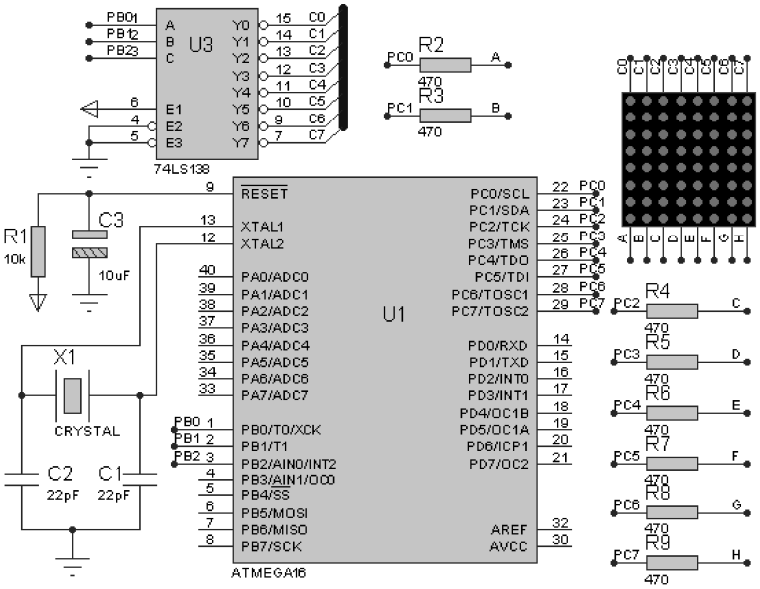


图 9-12　一个 8×8 点阵字符串显示（二）电路图

软件设计

由于每个字符有 8 个段码值，该字符串也有 15 个字符，所以该字符串库中有 8×15 共 120 个段码值。使用 PCtoLCD2002 字模软件，选择点阵格式为共阴极、取模方式为行列式、取模方向为低位在前，可生成这 15 个字符的字模段码值。程序流程图与图 9-10 基本相同。

源程序

```
/************************************************************
  文件名:          一个 8×8LED 点阵字符串显示（二）.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
  说明:            在 8×8LED 点阵上显示字符串  www.phei.com.cn
  取模方法:        宋体、共阴极、取模式方式为行列式、低位在前
************************************************************/
#include "mega16.h"
#define   uchar unsigned char
#define   uint unsigned   int
#define   CS1              PORTB.0                    //CS1 片选
#define   CS2              PORTB.1                    //CS2 片选
#define   CS3              PORTB.2                    //CS3 片选
#define   CS               PORTB
#define   LED               PORTC
uint   j,i,q=0,t=0;
flash uchar tab1[]={
      0x00,0x00,0x00,0xEB,0x49,0x55,0x22,0x00,        /*"w",0*/
      0x00,0x00,0x00,0xEB,0x49,0x55,0x22,0x00,        /*"w",1*/
      0x00,0x00,0x00,0xEB,0x49,0x55,0x22,0x00,        /*"w",2*/
      0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x06,0x00,   /*".",3*/
      0x00,0x00,0x00,0x3F,0x42,0x42,0x3E,0x07,        /*"p",4*/
      0x00,0x00,0x03,0x7E,0x42,0x42,0xE7,0x00,        /*"h",5*/
      0x00,0x00,0x00,0x3C,0x7E,0x02,0x7C,0x00,        /*"e",6*/
      0x00,0x00,0x0C,0x0E,0x08,0x08,0x3E,0x00,        /*"i",7*/
      0x00,0x00,0x00,0x00,0x00,0x00,0x06,0x00,        /*".",8*/
      0x00,0x00,0x00,0x7C,0x02,0x02,0x7C,0x00,        /*"c",9*/
      0x00,0x00,0x00,0x3C,0x42,0x42,0x3C,0x00,        /*"o",10*/
      0x00,0x00,0x00,0xFF,0x92,0x92,0xB7,0x00,        /*"m",11*/
      0x00,0x00,0x00,0x00,0x00,0x00,0x06,0x00,        /*".",12*/
      0x00,0x00,0x00,0x7C,0x02,0x02,0x7C,0x00,        /*"c",13*/
      0x00,0x00,0x00,0x7F,0x42,0x42,0xE7,0x00};       /*"n",14*/
flash uchar tab2[]={0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07};   //扫描代码（使用 74LS138）
/*********************************************
未使用 74LS138 时改用下列扫描代码:
flash uchar tab2[]={0x00,0x01,0x02,0x04,0x08,0x10,0x20,0x40};
*********************************************/
void delay(uint k)
{
   uint m,n;
```

```
    for(m=0;m<k;m++)
      {
        for(n=0;n<1140;n++)
          {;}
      }
}
void display(void)
{
   for(i=0;i<15;i++)
      {
        for(j=q;j<8+q;j++)
          {
            CS=tab2[t++];
            LED=tab1[j];
            delay(4);
            if(t= =8)
                  {
                t=0;
             }
          }
      }
   q=q+8;
   if(q= =120)
      {
        q=0;
      }
}
void port_init(void)                                    //端口初始化
{
     DDRB=0xFF;
    PORTB=0xFF;
     DDRC=0xFF;
    PORTC=0xFF;
}
void main(void)
{
    port_init();
    while(1)
       {
         display();
       }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“一个 8×8 点阵字符串显示（二）.COFF”文件，然后在 Proteus ISIS 中打开已创建的一个 8×8 点阵字符串显示（二）电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试运行状态下，可看见 8×8 点阵显示字符串“www.phei.com.cn”，运行效果如图 9-13 所示。

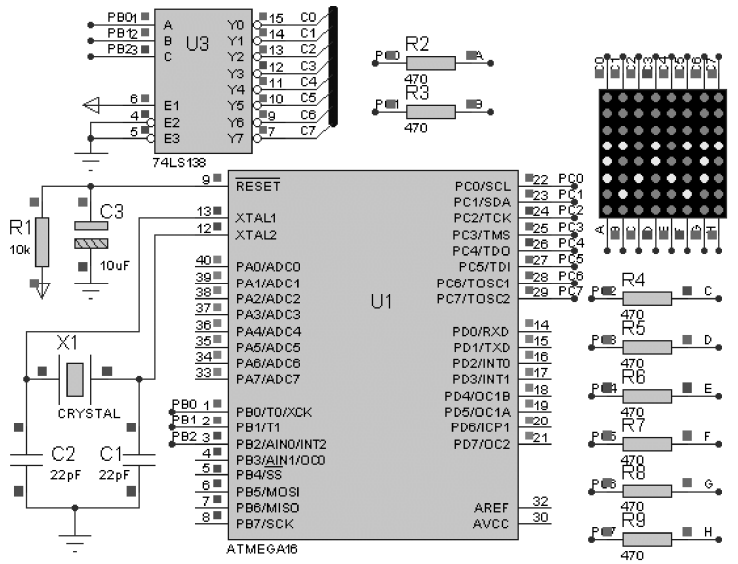


图 9-13　一个 8×8 点阵字符串（二）显示效果图

如果不使用 74LS138 进行位选时，P2 端口的 8 根口线要分别与 LED 点阵相应的位选端进行连接，且将程序的扫描代码改成如下即可，运行效果如图 9-14 所示。

```
flash uchar tab2[]={0x00,0x01,0x02,0x04,0x08,0x10,0x20,0x40};
```

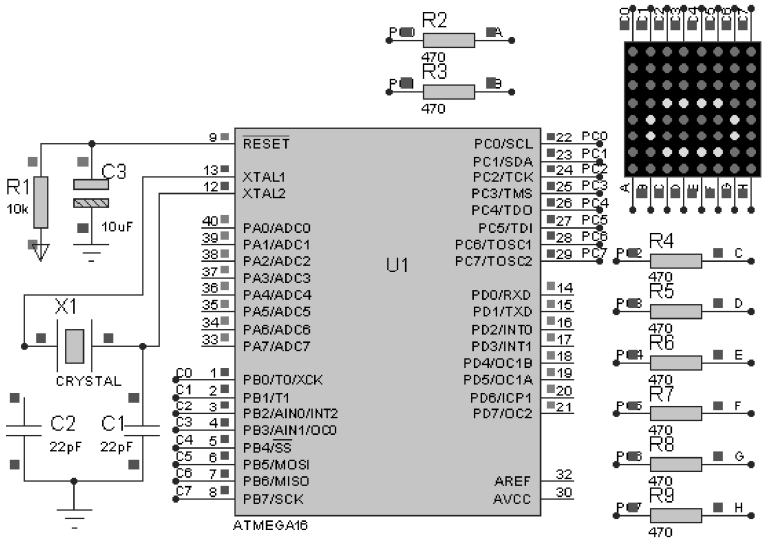


图 9-14　未使用 74LS138 仿真效果图

任务 4　两个 8×8 LED 点阵字符串显示

1．两个 8×8 LED 点阵字符串显示（一）

采用逐列式的取模方式，在两个 8×8 LED 点阵中显示字符串“ATmega16”。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 9-3 所列的元器件，并完成图 9-15 所示的硬件电路图设计。

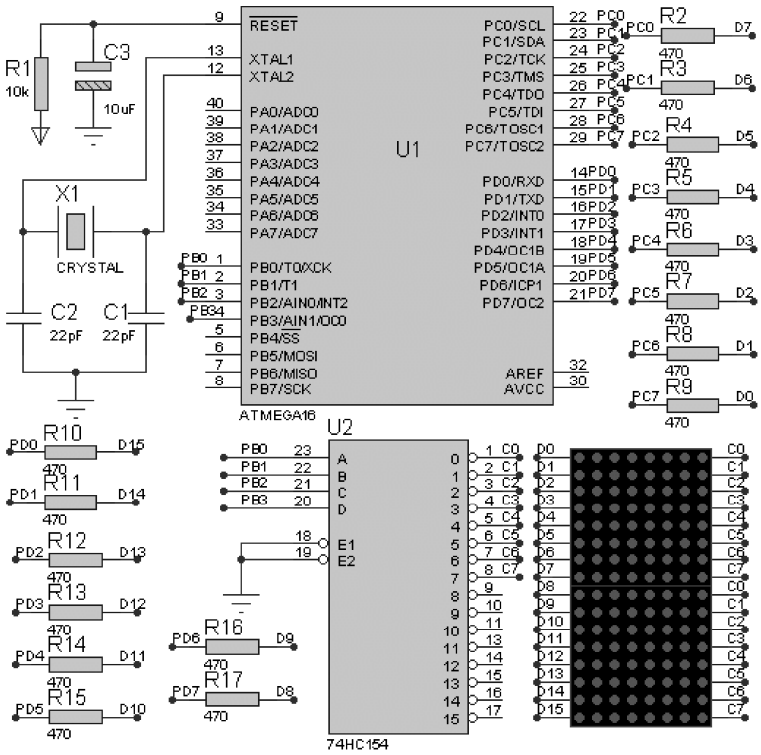


图 9-15　两个 8×8 LED 点阵字符串显示（一）电路图

表 9-3　两个 8×8 点阵字符串显示（一）所使用的元器件

单片机 ATmega16	电解电容 CAP-ELEC 10uF	瓷片电容 CAP22pF	电阻 RES
晶振 CRYSTAL 8MHz	点阵 MATRIX-8X8-RED	4-16 线译码器 74LS154	

软件设计

两个 8×8LED 点阵可构成一个 8×16 的 LED 点阵，每个字符由上半部分和下半部分构成，即 1 个段选位对应两个段码值。在某一时刻只能显示一个字符，要显示字符串，必须在显示完一个字符后接着显示下一个字符，因此需建立一个字符库。由于每个字符有 16 个段码值，该字符串有 8 个字符，所以该字符串库中有 16×8 共 128 个段码值。使用 PCtoLCD2002 字模软件，选择点阵格式为共阴极、取模方式为逐列式、取模方向为高位在前，可生成这 8 个字符的字模段码值。

8×16 共阴极 LED 点阵显示字符串“ATmega16”，可以通过建立一个数据表格的形式进行，首先位选 1 有效，将段码值 0x00 送给 PORTC 端口、段码值 0x00 送给 PORTD 端口以驱动相应段点亮，再然后位选 2 有效，将段码值 0x00 送给 PORTC 端口、段码值 0x7C 送给

PORTD 端口以驱动相应段点亮……如此进行，直到送完 16 个段码，就可显示“A”，然后再进行字符“T”的显示……每个字符的显示与字符“A”的显示过程相同，只是段码值不同而已。由于段码值较多，可采用一个结束码来进行判断是否字符串显示完，若字符串已显示完，又从头开始，因此该程序流程图如图 9-16 所示。

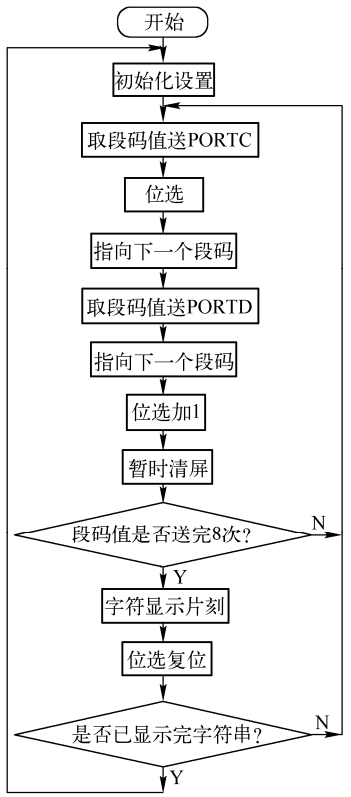


图 9-16 两个 8×8 LED 点阵字符串显示流程图

源程序

```
/************************************************************
  文件名:          两个 8×8LED 点阵字符串显示（一）.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
  说明:            在 8×8LED 点阵上显示字符串 ATmega16
  取模方法:        宋体、共阴极、取模式方式为列行式、高位在前
************************************************************/
#include "mega16.h"

#define   uchar unsigned char
#define   uint unsigned   int
#define   CS1             PORTB.0        //CS1 片选
#define   CS2             PORTB.1        //CS2 片选
#define   CS3             PORTB.2        //CS3 片选
#define   CS              PORTB
#define   LED1            PORTC
#define   LED2            PORTD
uint i,j,q=0,t=0;
flash uchar tab1[]={
        0x00,0x00,0x00,0x7C,0x07,0xA0,0x18,0x20,
        0x0F,0x20,0x00,0xF0,0x00,0x0C,0x00,0x00,  /*"A",0*/
        0x0C,0x00,0x10,0x00,0x10,0x00,0x1F,0xFC,
        0x10,0x04,0x10,0x00,0x18,0x00,0x00,0x00,  /*"T",1*/
        0x00,0x00,0x01,0xFC,0x01,0x00,0x01,0xFC,
        0x01,0x04,0x01,0x00,0x01,0xFC,0x00,0x00,  /*"m",2*/
        0x00,0x00,0x00,0xF8,0x01,0x24,0x01,0x24,
        0x01,0x24,0x01,0x24,0x00,0xE8,0x00,0x00,  /*"e",3*/
        0x00,0x00,0x00,0xFF,0x01,0x19,0x01,0x19,
        0x01,0x15,0x01,0xE5,0x01,0x07,0x00,0x00,  /*"g",4*/
        0x00,0x00,0x00,0x9C,0x01,0x24,0x01,0x24,
        0x01,0x24,0x01,0xF8,0x00,0x04,0x00,0x08,  /*"a",5*/
        0x00,0x00,0x00,0x00,0x08,0x00,0x0F,0xFC,
        0x1F,0xFC,0x00,0x00,0x00,0x00,0x00,0x00,  /*"1",6*/
        0x00,0x00,0x07,0xF8,0x08,0x84,0x11,0x04,
        0x11,0x04,0x19,0x04,0x08,0xF8,0x00,0x00};/*"6",7*/
flash uchar tab2[]={0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07};     //扫描代码
void delay(uint k)
{
   uint m,n;
   for(m=0;m<k;m++)
      {
         for(n=0;n<1140;n++)
```

```
            {;}
        }
    }
    void display(void)
    {
       for(i=0;i<15;i++)
          {
             for(j=q;j<16+q;j=j+2)
                {
                   CS=tab2[t++];
                   LED1=tab1[j];
                   LED2=tab1[j+1];
                   delay(4);
                   if(t= =8)
                     {
                        t=0;
                     }
                }
          }
     q=q+16;
     if(q= =128)
        {
           q=0;
        }
    }
    void port_init(void)                          //端口初始化
    {
         DDRB=0xFF;
        PORTB=0xFF;
         DDRC=0xFF;
        PORTC=0xFF;
         DDRD=0xFF;
        PORTD=0xFF;
    }
    void main(void)
    {
        port_init();
        while(1)
           {
              display();
           }
    }
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“两个 8×8 点阵字符串显示（一）.COFF”文件，然后在 Proteus ISIS 中打开已创建的两个 8×8 点阵字符串显示（一）电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试运行状态下，可看见两个 8×8 点阵显示字符串“ATmega16”，运行效果如图 9-17 所示。

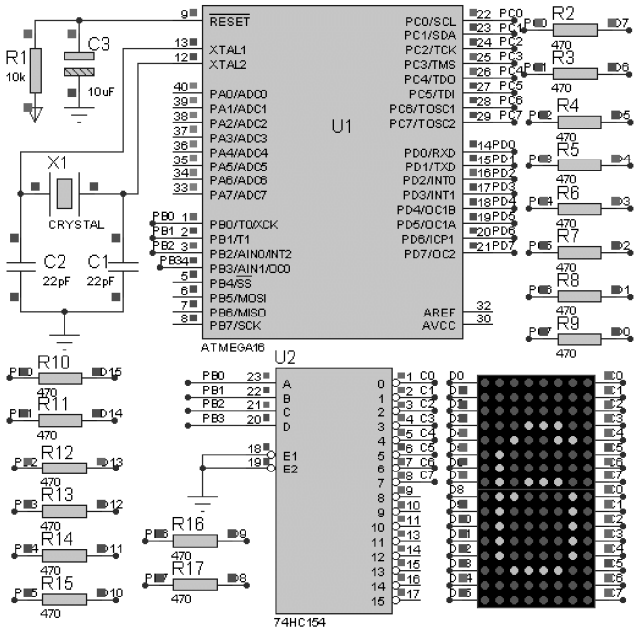


图 9-17　两个 8×8 点阵字符串显示“STC8951”仿真效果图

2．两个 8×8 LED 点阵字符串显示（二）

采用另一种方法使用两个 8×8 LED 点阵中显示字符串“CodeVision”。



硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 9-3 所列的元器件，并完成图 9-18 所示的硬件电路图设计。

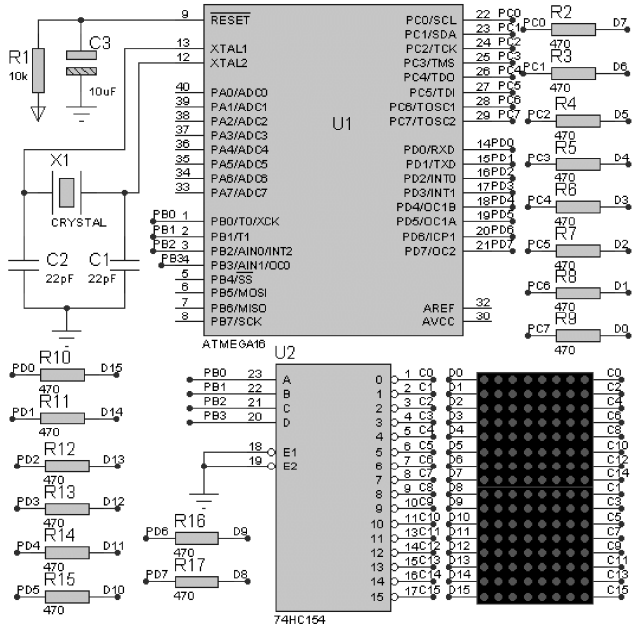


图 9-18　两个 8×8 LED 点阵字符串显示（二）电路图

软件设计

两个 8×8 LED 点阵中显示字符串“CodeVision”，该字符串有 10 个字符，由于每个字符有 16 个段码值，所以该字符串库中有 16×10 共 160 个段码值。使用 PCtoLCD2002 字模软件，选择点阵格式为共阴极、取模方式为逐列式、取模方向为高位在前，可生成这 10 个字符的字模段码值。其程序流程图与图 9-16 也类似，在此不再赘述。

源程序

```
/*************************************************************
  文件名:          两个 8×8LED 点阵字符串显示（二）.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
  说明:            在 8×8LED 点阵上显示字符串 CodeVision
  取模方法:        微软雅黑、共阴极、取模式方式为列行式、高位在前
*************************************************************/
#include "mega16.h"
#define   uchar unsigned char
#define   uint unsigned   int
#define   CS1              PORTB.0                    //CS1 片选
#define   CS2              PORTB.1                    //CS2 片选
#define   CS3              PORTB.2                    //CS3 片选
#define   CS4              PORTB.3                    //CS4 片选
#define   CS               PORTB
#define   LED1             PORTC
#define   LED2             PORTD
uint i=0,j,q=0,t=0;
flash uchar tab1[]={
        0x00,0x00,0x03,0xC0,0x04,0x20,0x0C,0x10,
        0x08,0x10,0x08,0x10,0x08,0x10,0x08,0x20,          /*"C",0*/
        0x00,0x00,0x00,0xE0,0x01,0xB0,0x01,0x10,
        0x01,0x10,0x01,0x10,0x01,0x10,0x00,0xE0,          /*"o",1*/
        0x00,0x00,0x00,0xE0,0x01,0xB0,0x01,0x10,
        0x01,0x10,0x01,0x10,0x01,0x20,0x0F,0xF0,          /*"d",2*/
        0x00,0x00,0x00,0xE0,0x01,0x50,0x01,0x50,
        0x01,0x50,0x01,0x50,0x00,0xD0,0x00,0x00,          /*"e",3*/
        0x08,0x00,0x0E,0x00,0x07,0x80,0x01,0xE0,
        0x00,0x30,0x00,0x30,0x01,0xE0,0x07,0x00,          /*"V",4*/
        0x00,0x00,0x09,0xF0,0x08,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,          /*"i",5*/
        0x00,0x00,0x01,0x90,0x01,0x50,0x01,0x50,
        0x01,0x20,0x00,0x00,0x00,0x00,0x00,0x00,          /*"s",6*/
        0x00,0x00,0x09,0xF0,0x08,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,          /*"i",7*/
        0x00,0x00,0x00,0xE0,0x01,0xB0,0x01,0x10,
        0x01,0x10,0x01,0x10,0x01,0x10,0x00,0xE0,          /*"o",8*/
        0x00,0x00,0x01,0xF0,0x00,0x80,0x01,0x00,
```

```
        0x01,0x00,0x01,0x00,0x00,0xF0,0x00,0x00};                    /*"n",9*/
flash uchar tab2[]={0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,         //扫描代码
                    0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F};
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void display(void)
{
  for(i=0;i<15;i++)                                                  //控制每一个字符显示的时间
      for(j=q;j<16+q;j++)
          {
           if(t%2= =0)
               {
                    CS=tab2[t];
               LED1=tab1[j];
                delay(2);
                }
       else
            {
                   CS=tab2[t];
              LED2=tab1[j];
                   delay(2);
                }
             t++;
        if(t= =16)
          t=0;
          }
        q=q+16;                                                      //显示下一个字符
        if(q= =160)
          {
            q=0;
          }
}
void port_init(void)                                                 //端口初始化
{
    DDRB=0xFF;
   PORTB=0xFF;
    DDRC=0xFF;
   PORTC=0xFF;
    DDRD=0xFF;
   PORTD=0xFF;
}
void main(void)
{
```

```
    port_init();
    while(1)
      {
        display();
      }
}
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“两个 8×8 点阵字符串显示（二）.COFF”文件，然后在 Proteus ISIS 中打开已创建的两个 8×8 点阵字符串显示（二）电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试运行状态下，可看见两个 8×8 点阵显示字符串“Codevision”，运行效果如图 9-19 所示。

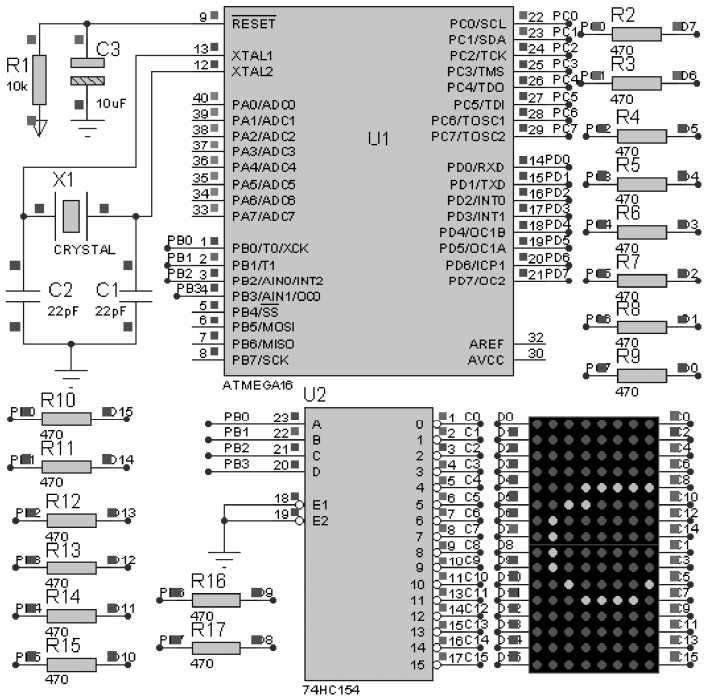


图 9-19　两个 8×8 点阵字符串显示“CodeVision”仿真效果图

任务 5　两个 8×8 LED 点阵滚动显示

使用两个 8×8 共阴极 LED 点阵滚动显示字符串“Proteus7.10”。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 9-3 所列的元器件，并完成图 9-20 所示的硬件电路图设计。

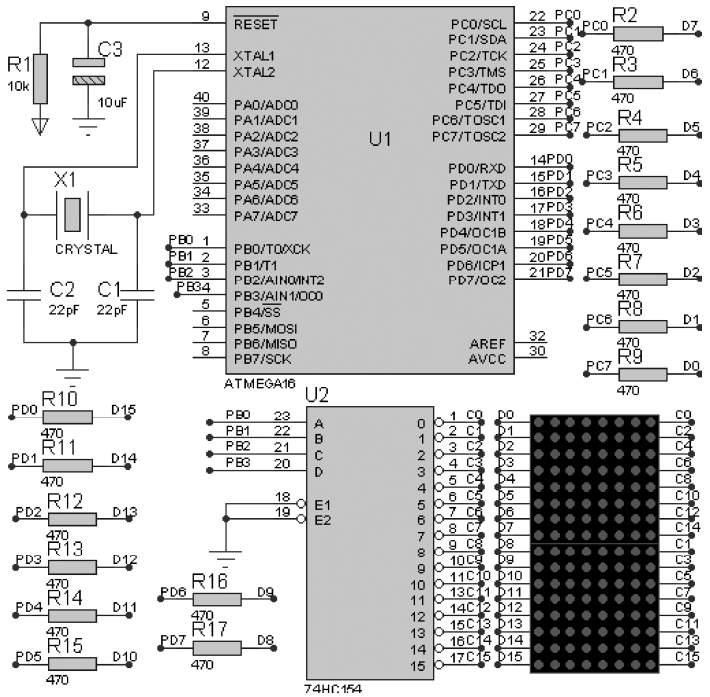


图 9-20　两个 8×8 LED 点阵滚动显示电路图

软件设计

两个 8×8 LED 点阵滚动显示字符串“Proteus7.10”，该字符串有 11 个字符，由于每个字符有 16 个段码值，所以该字符串库中有 16×11 共 176 个段码值。使用 PCtoLCD2002 字模软件，选择点阵格式为共阴极、取模方式为逐列式、取模方向为高位在前，可生成这 11 个字符的字模段码值。

两个 8×8 点阵滚动显示是在任务 4 的基础上进行移动显示，只不过要进行移动显示时，将“q=q+16”改为“q=q+2”或“q=q+4”即可（q 加上一个大于 0 且小于 16 的偶数），其程序流程图请参阅图 9-16。

源程序

```
/************************************************************
  文件名:          两个 8×8LED 点阵滚动显示.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
  说明:            在 8×8LED 点阵上显示字符串 Proteus7
  取模方法:        微软雅黑、共阴极、取模式方式为列行式、高位在前
************************************************************/
#include "mega16.h"
#define   uchar unsigned char
#define   uint unsigned   int
#define   CS1              PORTB.0                    //CS1 片选
#define   CS2              PORTB.1                    //CS2 片选
```

```
#define   CS3              PORTB.2                                  //CS3 片选
#define   CS4              PORTB.3                                  //CS4 片选
#define   CS               PORTB
#define   LED1             PORTC
#define   LED2             PORTD
uint j=0,r,q=0,t=0;
flash uchar tab1[]={
          0x00,0x00,0x0F,0xF0,0x08,0x80,0x08,0x80,
          0x08,0x80,0x08,0x80,0x0D,0x00,0x07,0x00,                 /*"P",0*/
          0x00,0x00,0x01,0xF0,0x00,0x80,0x01,0x00,
          0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x00,                 /*"r",1*/
          0x00,0x00,0x00,0xE0,0x01,0xB0,0x01,0x10,
          0x01,0x10,0x01,0x10,0x01,0x10,0x00,0xE0,                 /*"o",2*/
          0x00,0x00,0x01,0x00,0x07,0xF0,0x01,0x10,
          0x01,0x10,0x00,0x00,0x00,0x00,0x00,0x00,                 /*"t",3*/
          0x00,0x00,0x00,0xE0,0x01,0x50,0x01,0x50,
          0x01,0x50,0x01,0x50,0x00,0xD0,0x00,0x00,                 /*"e",4*/
          0x00,0x00,0x01,0xE0,0x00,0x10,0x00,0x10,
          0x00,0x10,0x00,0x10,0x01,0xF0,0x00,0x00,                 /*"u",5*/
          0x00,0x00,0x01,0x90,0x01,0x50,0x01,0x50,
          0x01,0x20,0x00,0x00,0x00,0x00,0x00,0x00,                 /*"s",6*/
          0x00,0x00,0x08,0x00,0x08,0x10,0x08,0xE0,
          0x0B,0x80,0x0E,0x00,0x08,0x00,0x00,0x00,                 /*"7",7*/
          0x00,0x00,0x00,0x10,0x00,0x10,0x00,0x00,
          0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,                 /*".",8*/
          0x00,0x00,0x00,0x00,0x04,0x10,0x08,0x10,
          0x0F,0xF0,0x00,0x10,0x00,0x10,0x00,0x00,                 /*"1",9*/
          0x00,0x00,0x03,0xE0,0x0C,0x30,0x08,0x10,
          0x08,0x10,0x0C,0x30,0x07,0xC0,0x00,0x00};                /*"0",10*/
flash uchar tab2[]={0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,       //扫描代码
                    0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F};
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void display(void)
{
  for(r=0;r<40;r++)                                                //控制每一个字符显示的时间
    {
      for(j=q;j<16+q;j++)
          {
            if(t%2= =0)
                {
```

```
                CS=tab2[t];
                LED1=tab1[j];
                delay(4);
            }
          else
          {
                CS=tab2[t];
                LED2=tab1[j];
                delay(4);
            }
          t++;
        if(t= =16)
          {
            t=0;
          }
      }
     q=q+2;                                          //显示下一个字符
      if(q= =176)
        {
          q=0;
        }
   }
}
void port_init(void)                                 //端口初始化
{
    DDRB=0xFF;
   PORTB=0xFF;
    DDRC=0xFF;
   PORTC=0xFF;
    DDRD=0xFF;
   PORTD=0xFF;
}
void main(void)
{
   port_init();
   while(1)
     {
       display();
     }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“两个 8×8 点阵滚动显示. COFF”文件，然后在 Proteus ISIS 中打开已创建的两个 8×8 点阵滚动显示电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试运行状态下，可看见两个 8×8 点阵滚动显示字符串“Proteus7.10”，运行效果如图 9-21 所示。

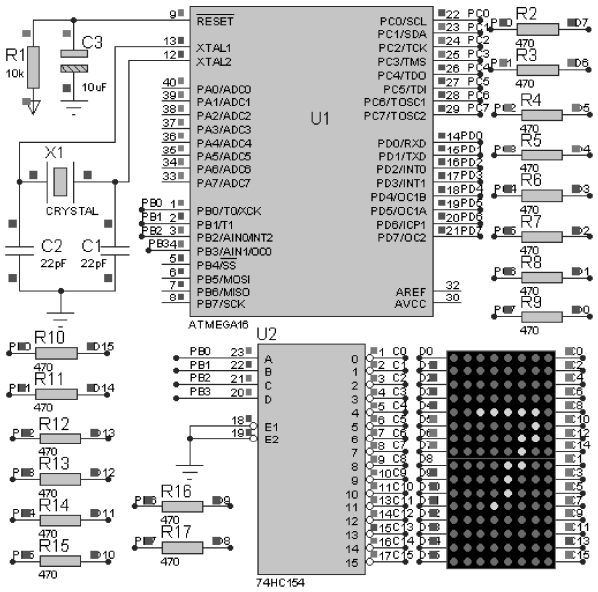


图 9-21　两个 8×8 点阵滚动显示“Proteus7.10”仿真效果图

任务 6　一个 16×16 LED 点阵汉字显示

使用一个 16×16 共阴极 LED 点阵显示汉字字符串“基于 Proteus 的 ATmega16 单片机设计”。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 9-4 所列的元器件，并完成图 9-22 所示的硬件电路图设计。

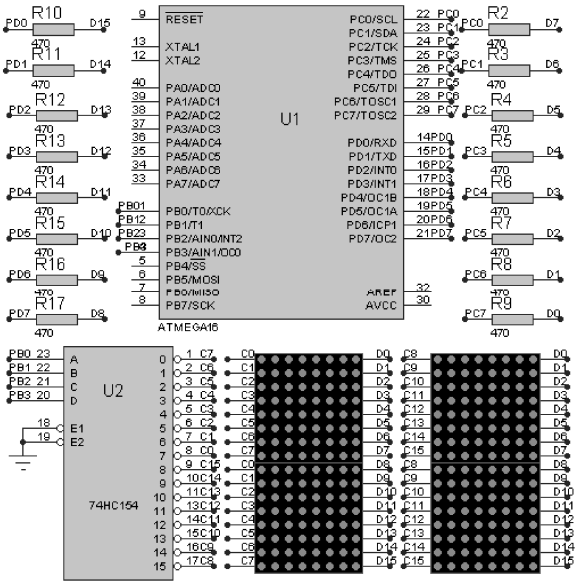


图 9-22　一个 16×16 LED 点阵汉字显示（一）电路图

注意:

图中单片机的晶振电路、复位电路等部分没有绘制出。

表 9-4　一个 16×16 LED 点阵汉字显示所使用的元器件

单片机 ATmega16	电解电容 CAP-ELEC 10uF	瓷片电容 CAP22pF	电阻 RES
晶振 CRYSTAL 8MHz	点阵 MATRIX-8X8-GREEN	4-16 线译码器 74LS154	

软件设计

一个 16×16 共阴极 LED 点阵，实质上是由 4 个 8×8 点阵构成的，要显示字符串“基于 Proteus 的 ATmega16 单片机设计”，可以通过建立一个数据表格的形式进行。由于一个汉字占用 32 个段码，而一个字母占用 16 个段码，在建立一个汉字字符库时，应在合适的位置插入相应 0x00 代码进行的空白显示，否则达不到显示设计效果，因此该字符串库中有 512 个段码值。使用 PCtoLCD2002 字模软件，选择点阵格式为共阴极、取模方式为逐列式、取模方向为高位在前，字体为“黑体”，可生成“基于 Proteus 的 ATmega16 单片机设计”这几个字符的字模段码值。

首先位选 1 有效，将段码值 0x00 送给 PORTC 端口、0x00 送给 PORTD 端口以驱动相应段点亮，然后位选 2 有效，将段码值 0x00 送给 PORTC 端口、0x88 送给 PORTD 端口以驱动相应段点亮……如此进行，直到送完 32 个段码，就可显示“基”字，然后再进行字符“于”的显示……每个字符的显示与字符“于”的显示过程相同，只是段码值不同而已。片选位有 C0～C15，由单片机的 PORTB 口控制 74HC154 输出。因此其程序流程图如图 9-23 所示。

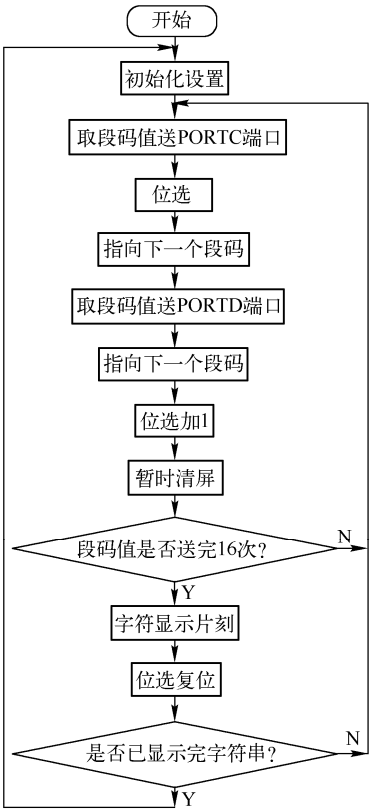


图 9-23　一个 16×16 点阵汉字显示流程图

源程序

```
/************************************************************
 文件名:          一个 16×16LED 点阵汉字显示.c
 单片机型号:      ATmega16
 晶振频率:        8.0MHz
 说明:            在 16×16LED 点阵上显示基于 Proteus 的 ATmega16 单片机设计
 取模方法:        微软雅黑、共阴极、取模式方式为逐列式、高位在前
************************************************************/
#include "mega16.h"
#define   uchar unsigned char
#define   uint unsigned   int
#define   CS1             PORTB.0                    //CS1 片选
#define   CS2             PORTB.1                    //CS2 片选
#define   CS3             PORTB.2                    //CS3 片选
#define   CS4             PORTB.3                    //CS4 片选
#define   CS              PORTB
#define   LED1            PORTC
#define   LED2            PORTD
uint j=0,r,q=0,t=0;
flash uchar tab1[]={
        0x00,0x00,0x00,0x88,0x10,0x9A,0x10,0x92,
        0x7F,0xAA,0x7F,0xCA,0x16,0x8A,0x16,0xBE,
        0x16,0xBE,0x16,0x8A,0x7F,0xCA,0x10,0xA2,
        0x10,0x92,0x10,0x9A,0x00,0x88,0x00,0x00,          /*"基",0*/
        0x00,0x00,0x01,0x00,0x21,0x00,0x21,0x00,
        0x21,0x00,0x21,0x02,0x21,0x02,0x3F,0xFE,
        0x3F,0xFC,0x21,0x00,0x21,0x00,0x21,0x00,
        0x21,0x00,0x01,0x00,0x01,0x00,0x00,0x00,          /*"于",1*/
        0x00,0x00,0x1F,0xFC,0x10,0xC0,0x10,0xC0,
        0x10,0x80,0x18,0x80,0x0F,0x80,0x00,0x00,          /*"P",2*/
        0x00,0x00,0x00,0x00,0x01,0xFC,0x00,0x80,
        0x01,0x00,0x01,0x00,0x00,0x00,0x00,0x00,          /*"r",3*/
        0x00,0x00,0x00,0xF8,0x01,0x08,0x01,0x04,
        0x01,0x04,0x01,0x88,0x00,0xF8,0x00,0x00,          /*"o",4*/
        0x01,0x00,0x01,0x00,0x01,0x00,0x0F,0xF8,
        0x01,0x04,0x01,0x04,0x00,0x0C,0x00,0x00,          /*"t",5*/
        0x00,0x00,0x00,0xF8,0x01,0x48,0x01,0x44,
        0x01,0x44,0x01,0xCC,0x00,0xD8,0x00,0x00,          /*"e",6*/
        0x00,0x00,0x01,0xF8,0x00,0x0C,0x00,0x04,
        0x00,0x0C,0x00,0x18,0x01,0xFC,0x00,0x00,          /*"u",7*/
        0x00,0x00,0x00,0xD8,0x01,0xCC,0x01,0x24,
        0x01,0x24,0x01,0xAC,0x00,0x98,0x00,0x00,          /*"s",8*/
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
```

```
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,        /*" ",8*/
        0x00,0x00,0x00,0x00,0x0F,0xFE,0x18,0x84,
        0x78,0x84,0x08,0x84,0x0F,0xFE,0x01,0x00,
        0x07,0x00,0x3C,0x80,0x68,0xE2,0x08,0x22,
        0x08,0x02,0x0F,0xFE,0x0F,0x80,0x00,0x00,        /*"的",9*/
        0x00,0x04,0x00,0x7C,0x03,0xE0,0x1E,0x40,
        0x1E,0x40,0x03,0xE0,0x00,0x3C,0x00,0x04,        /*"A",10*/
        0x00,0x00,0x10,0x00,0x10,0x00,0x1F,0xFC,
        0x1F,0xFC,0x10,0x00,0x10,0x00,0x00,0x00,        /*"T",11*/
        0x01,0xFC,0x01,0xFC,0x01,0x00,0x01,0xFC,
        0x00,0xFC,0x01,0x00,0x01,0xFC,0x01,0xFC,        /*"m",12*/
        0x00,0x00,0x00,0xF8,0x01,0x48,0x01,0x44,
        0x01,0x44,0x01,0xCC,0x00,0xD8,0x00,0x00,        /*"e",13*/
        0x00,0x00,0x00,0xDF,0x01,0x29,0x01,0x29,
        0x01,0x29,0x01,0xED,0x01,0x06,0x00,0x00,        /*"g",14*/
        0x00,0x00,0x00,0xB8,0x01,0x24,0x01,0x24,
        0x01,0x4C,0x01,0xF8,0x00,0xFC,0x00,0x00,        /*"a",15*/
        0x00,0x00,0x06,0x00,0x04,0x00,0x0F,0xFC,
        0x1F,0xFC,0x00,0x00,0x00,0x00,0x00,0x00,        /*"1",16*/
        0x00,0x00,0x01,0xF8,0x03,0x8C,0x0F,0x04,
        0x19,0x04,0x01,0x0C,0x00,0xF8,0x00,0x60,        /*"6",17*/
        0x00,0x00,0x00,0x08,0x00,0x08,0x0F,0xC8,
        0x4B,0x48,0x7B,0x48,0x0B,0x48,0x0F,0xFF,
        0x0F,0xFF,0x1B,0x48,0x6B,0x48,0x0B,0x48,
        0x0F,0xC8,0x00,0x08,0x00,0x08,0x00,0x00,        /*"单",18*/
        0x00,0x00,0x00,0x02,0x00,0x0E,0x7F,0xF8,
        0x7F,0xC0,0x04,0x40,0x04,0x40,0x04,0x40,
        0x04,0x40,0x04,0x40,0x7C,0x40,0x04,0x7F,
        0x04,0x00,0x04,0x00,0x04,0x00,0x00,0x00,        /*"片",19*/
        0x00,0x00,0x08,0x30,0x09,0xC0,0x7F,0xFE,
        0x7F,0xFE,0x09,0xC2,0x08,0x46,0x00,0x1C,
        0x3F,0xF0,0x20,0x00,0x20,0x00,0x3F,0xFC,
        0x3F,0xFE,0x00,0x02,0x00,0x06,0x00,0x00,        /*"机",20*/
        0x00,0x00,0x02,0x00,0x62,0x00,0x33,0xFE,
        0x10,0x0D,0x03,0x19,0x06,0x82,0x3C,0xE2,
        0x20,0xB4,0x20,0x8C,0x20,0x9C,0x3E,0xB6,
        0x02,0xC2,0x02,0x02,0x02,0x03,0x00,0x00,        /*"设",21*/
        0x00,0x00,0x02,0x00,0x22,0x00,0x33,0xFE,
        0x18,0x0C,0x02,0x18,0x02,0x10,0x02,0x00,
        0x02,0x00,0x02,0x00,0x7F,0xFE,0x02,0x00,
        0x02,0x00,0x02,0x00,0x02,0x00,0x00,0x00};       /*"计",22*/
flash uchar tab2[]={0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,        //扫描代码
                    0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F};
void delay(uint k)
```

```
{
   uint m,n;
   for(m=0;m<k;m++)
      {
         for(n=0;n<114;n++)
           {;}
      }
}
void display(void)
{
   for(r=0;r<30;r++)                                    //控制每一个字符显示的时间
       {
                for(j=q;j<32+q;j++)                     //一个汉字数据占用 32B
                 {
                   CS=tab2[t];                          //扫描
                  LED1=tab1[j];                         //送数据
                  j++;
                  LED2=tab1[j];                         //送数据
                    delay(10);
                    t++;
                  if(t= =16)
                       {
                             t=0;
                        }
                   }
              }
       q=q+32;                                          //显示下一个字符
        if(q= =512)
          {
              q=0;
          }
}
void port_init(void)                                    //端口初始化
{
      DDRB=0xFF;
     PORTB=0xFF;
      DDRC=0xFF;
     PORTC=0xFF;
      DDRD=0xFF;
     PORTD=0xFF;
}
void main(void)
{
     port_init();
```

```
    while(1)
      {
        display();
      }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“一个 16×16LED 点阵汉字显示.COFF”文件，然后在 Proteus ISIS 中打开已创建的一个 16×16LED 点阵汉字显示电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试运行状态下，可看见一个 16×16LED 点阵汉字显示效果如图 9-24 所示。

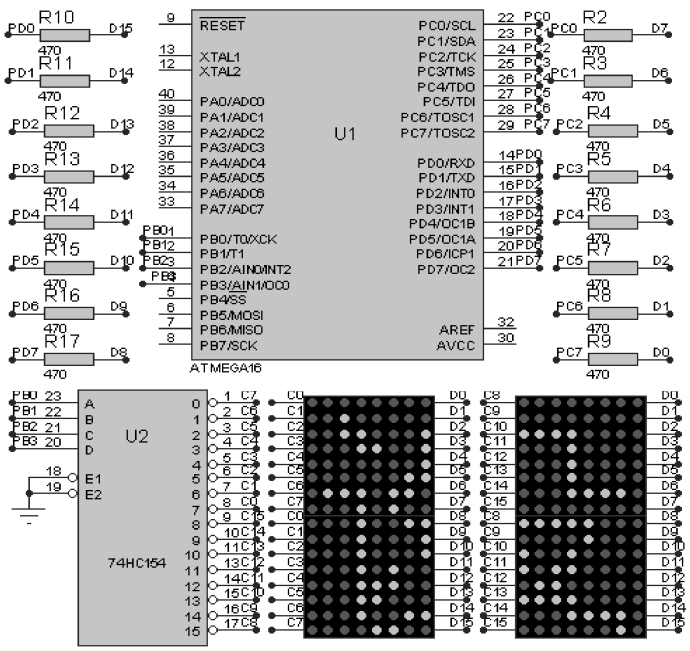


图 9-24　一个 16×16LED 点阵汉字显示仿真效果图

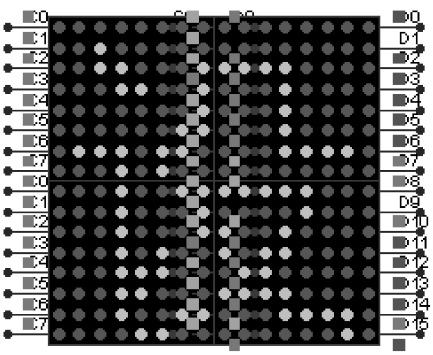


图 9-25　直接接合在一起的显示效果

图 9-24 中显示的汉字就像是两个半体，未形成一个完整的汉字。若直接将两者移动接合在一起时，显示汉字的效果如图 9-25 所示。在此图中会显示一些引脚电平，影响显示效果。

为隐藏一些引脚电平，达到较好的显示效果，先将 LED 点阵合并在一起，并全部选中，再执行菜单命令“system”→“Set Animation Options…”，打开“Animated Circuits Configuration”对话框，然后在“Animation Options”区域中取消“Show Logic State of Pins?”选项的选中状态，如图 9-26 所示。设置好后，重新单击按钮，运行效果如图 9-27 所示。

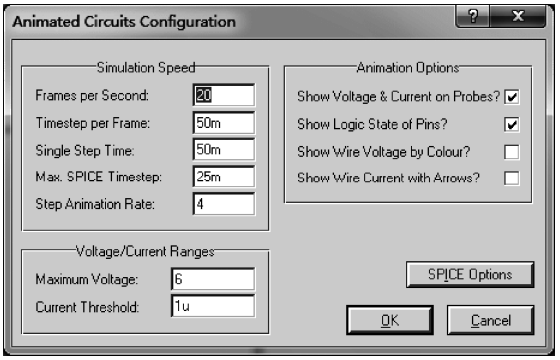


图 9-26　隐藏引脚逻辑状态

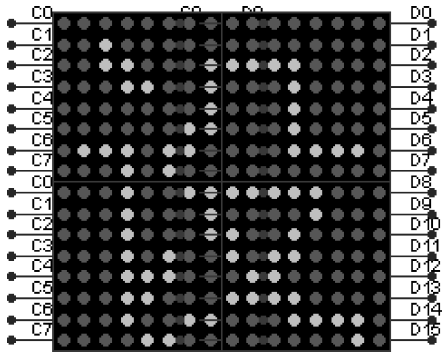


图 9-27　隐藏引脚电平后的显示效果

任务 7　一个 16×16 LED 点阵汉字移位显示

使用一个 16×16 共阴极 LED 点阵移位显示汉字字符串“全国电子协会主办”。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 9-4 所列的元器件，并完成图 9-28 所示的硬件电路图设计。

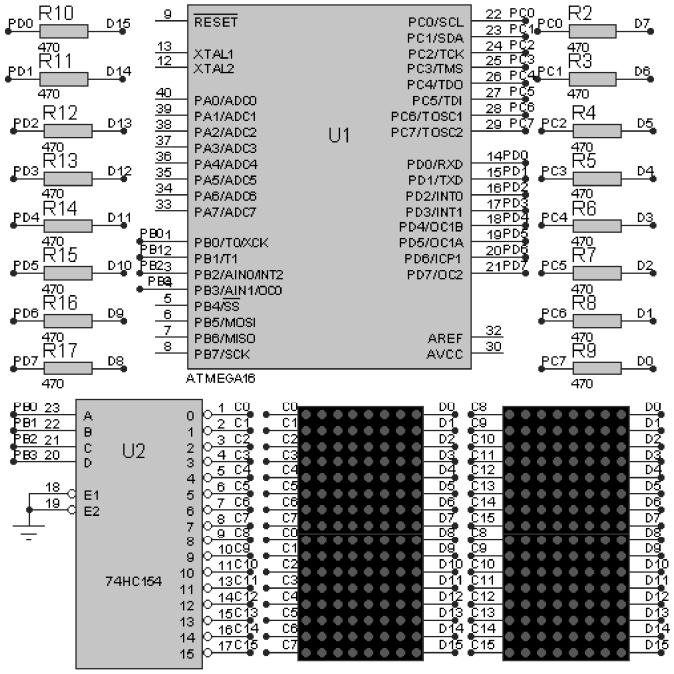


图 9-28　一个 16×16 LED 点阵汉字移位显示电路图

注意：

图中单片机的晶振电路、复位电路等部分没有绘制出。

软件设计

要实现一个 16×16 LED 点阵汉字移位显示，可以将任务 6 中的源程序进行修改即可，因此其程序流程图请参见图 9-23。

源程序

```
/************************************************************
  文件名:          一个 16×16LED 点阵汉字移位显示.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
  说明:            在 16×16LED 点阵上显示  全国电子协会主办
  取模方法:        黑体、共阴极、取模式方式为逐列式、高位在前
************************************************************/
#include "mega16.h"
#define   uchar unsigned char
#define   uint unsigned   int
#define   CS1             PORTB.0                    //CS1 片选
#define   CS2             PORTB.1                    //CS2 片选
#define   CS3             PORTB.2                    //CS3 片选
#define   CS4             PORTB.3                    //CS4 片选
#define   CS              PORTB
#define   LED1            PORTC
#define   LED2            PORTD
uint j=0,r,q=0,t=0;
flash uchar tab1[]={
      0x00,0x00,0x06,0x06,0x06,0x06,0x0C,0x66,
      0x1E,0x66,0x36,0x66,0x66,0x66,0xC7,0xFE,
      0xE7,0xFE,0x36,0x66,0x1E,0x66,0x0E,0x66,
      0x0C,0x66,0x06,0x06,0x06,0x06,0x00,0x00,0x00,          /*"全",0*/
      0x00,0x00,0x7F,0xFE,0x7F,0xFE,0x60,0x06,
      0x6D,0x36,0x6D,0x36,0x6D,0x36,0x6F,0xF6,
      0x6F,0xF6,0x6D,0x36,0x6D,0x76,0x60,0x36,
      0x7F,0xFE,0x7F,0xFE,0x00,0x00,0x00,0x00,          /*"国",1*/
      0x00,0x00,0x00,0x00,0x3F,0xF0,0x3F,0xF0,
      0x32,0x60,0x32,0x60,0xFF,0xFC,0xFF,0xFE,
      0x32,0x66,0x32,0x66,0x32,0x66,0x3F,0xE6,
      0x3F,0xE6,0x00,0x1E,0x00,0x1C,0x00,0x00,          /*"电",2*/
      0x03,0x00,0x03,0x00,0xC3,0x00,0xC3,0x00,
      0xC3,0x00,0xC3,0x04,0xC3,0x06,0xCF,0xFE,
      0xCF,0xFC,0xDB,0x00,0xF3,0x00,0xE3,0x00,
      0xC3,0x00,0x03,0x00,0x03,0x00,0x00,0x00,          /*"子",3*/
      0x0C,0x00,0x0C,0x00,0xFF,0xFE,0xFF,0xFE,
      0x0C,0x00,0x01,0xC6,0x19,0x9C,0x18,0x78,
      0xFF,0xE0,0xFF,0x84,0x18,0x06,0x1F,0xFE,
      0x1F,0xFC,0x01,0xC0,0x01,0xC0,0x00,0x00,          /*"协",4*/
      0x00,0x00,0x06,0xC0,0x06,0xC6,0x0C,0xCE,
      0x18,0xDE,0x36,0xF6,0xE6,0xE4,0xE6,0xC4,
      0x36,0xC4,0x1E,0xD4,0x08,0xDC,0x0C,0xCE,
      0x06,0xC6,0x06,0xC0,0x04,0x00,0x00,0x00,          /*"会",5*/
      0x00,0x00,0x18,0x06,0x18,0xC6,0x18,0xC6,
```

```
        0x18,0xC6,0x18,0xC6,0xD8,0xC6,0x7F,0xFE,
        0x3F,0xFE,0x18,0xC6,0x18,0xC6,0x18,0xC6,
        0x18,0xC6,0x18,0x06,0x00,0x00,0x00,0x00,                    /*"主",6*/
        0x00,0x40,0x00,0xC2,0x1B,0x86,0x1A,0x0C,
        0x18,0x18,0x18,0xF0,0xFF,0xC0,0xFF,0x06,
        0x18,0x06,0x18,0x06,0x1F,0xFC,0x1F,0xF8,
        0x01,0x80,0x01,0xE0,0x00,0x60,0x00,0x00};                   /*"办",7*/
flash uchar tab2[]={0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00,        //扫描代码
                    0x0F,0x0E,0x0D,0x0C,0x0B,0x0A,0x09,0x08};
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<114;n++)
        {;}
    }
}
void display(void)
{
  for(r=0;r<20;r++)                                                 //控制每一个字符显示的时间
    {
      for(j=q;j<32+q;j++)                                           //一个汉字数据占用 32B
        {
          CS=tab2[t];                                               //扫描
          LED1=tab1[j];                                             //送数据
          j++;
          LED2=tab1[j];                                             //送数据
          delay(5);
          t++;
          if(t= =16)
                {
                t=0;
            }
        }
    }
    q=q+8;                                                          //显示下一个字符
    if(q= =256)
      {
        q=0;
      }
}
void port_init(void)                                                //端口初始化
{
     DDRB=0xFF;
    PORTB=0xFF;
     DDRC=0xFF;
    PORTC=0xFF;
     DDRD=0xFF;
    PORTD=0xFF;
}
void main(void)
```

```
{
    port_init();
    while(1)
      {
        display();
      }
}
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“一个 16×16 LED 点阵汉字移位显示.COFF”文件，然后在 Proteus ISIS 中打开已创建的一个 16×16 LED 点阵汉字移位显示电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试运行状态下，可看见一个 16×16LED 点阵汉字显示效果，如图 9-29 所示。

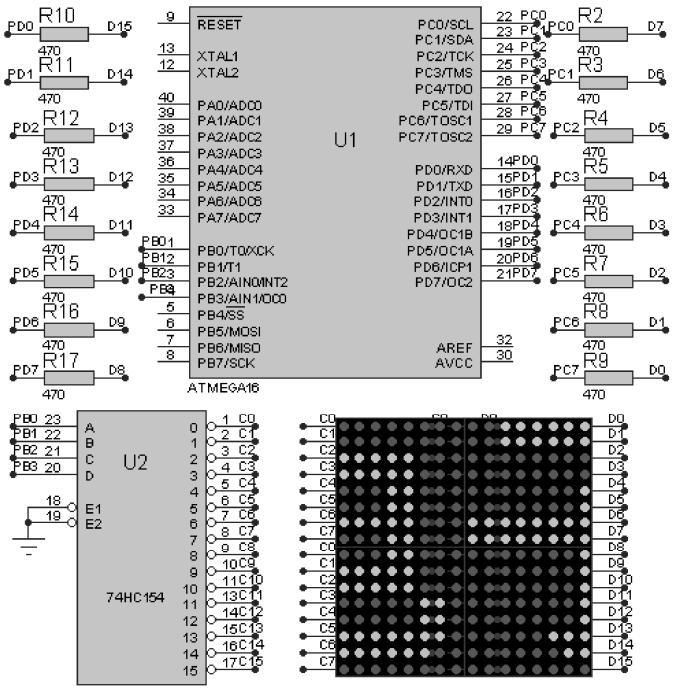


图 9-29　一个 16×16 LED 点阵汉字移位显示仿真效果图

任务 8　两个 16×16 LED 点阵汉字显示

使用两个 16×16 共阴极 LED 点阵显示字符串“电子工业出版社”。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 9-4 所列的元器件，并完成图 9-30 所示的硬件电路图设计。

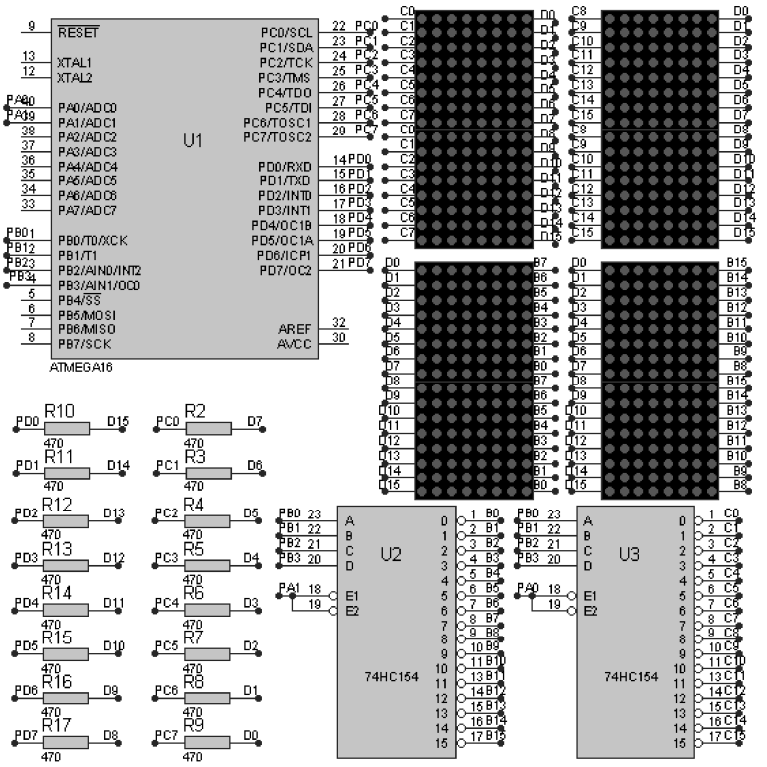


图 9-30　两个 16×16 LED 点阵汉字显示电路

> 注意:
> 图中单片机的晶振电路、复位电路等部分没有绘制出。

软件设计

两个 16×16 LED 点阵可构成一个 16×32 的 LED 点阵，实质上是由 8 个 8×8 点阵构成的，如图 9-30 所示。8 个 8×8 点阵可由单片机 P0 端口和 PORTD 端口输出段码值，片选位由两个 74HC154 控制，单片机的 PORTB 口输出 LED 的片选值，PA0(SE0)和 PA1(SE1)选择相应的 74HC154。

每个字符由上半部分和下半部分构成，即 1 个段选位对应 4 个段码值。16×32 共阴极 LED 点阵显示字符串“电子工业出版社”，可以通过建立一个数据表格的形式进行。使用 16×32 点阵时，一个汉字占用 64 个段码，显示的内容共有 7 个汉字，所以字符串库中有 448 个段码值。使用 PCtoLCD2002 字模软件，选择点阵格式为共阴极、取模方式为逐列式、取模方向为高位在前，字体为“黑体”，可生成“电子工业出版社”这 7 个汉字的字模段码值。

首先设置 SE0=0，SE1=1 选中 U3 的 74HC154，使 U3 位选 1 有效，将段码值 0x00 送给端口 PC、段码值 0x00 送给 PD 端口以驱动相应段点亮，再设置 SE0=1，SE1=0 选中 U2 的 74HC154，使 U2 位选 1 有效，将段码值 0x00 送给 PC 端口、段码值 0x00 送给 PD 端口以驱动相应段点亮。然后设置 SE0=0，SE1=1 选中 U3 的 74HC154，使 U3 位选 2 有效，将段码值 0x00 送给 PC 端口、段码值 0x00 送给 PD 端口以驱动相应段点亮……如此进行，直到送完 64 个段码，就可显示“电”，然后再进行字符“子”的显示……每个字符的显示与字

符“电”的显示过程相同，只是段码值不同而已。其程序流程图如图 9-31 所示。

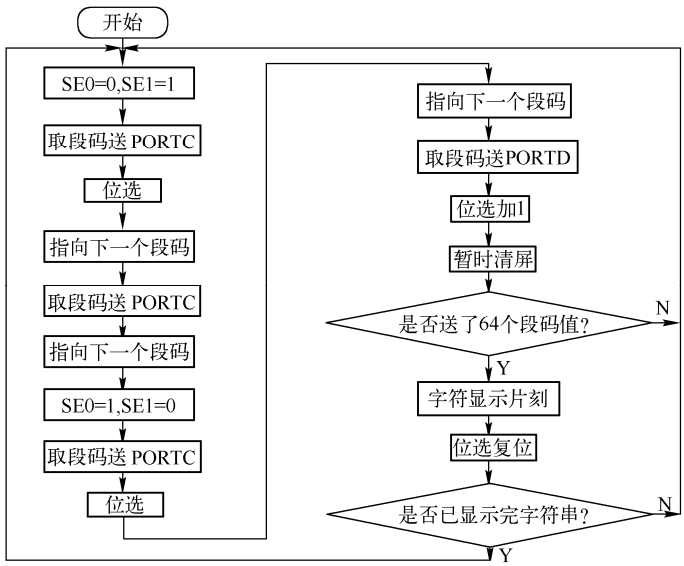


图 9-31　两个 16×16 LED 点阵汉字显示程序流程图

源程序

```
/************************************************************
 文件名:            两个 16×16LED 点阵汉字显示.c
 单片机型号:        ATmega16
 晶振频率:          8.0MHz
 说明:              在 16×16LED 点阵上显示   电子工业出版社
 取模方法:          黑体、共阴极、取模式方式为逐列式、高位在前
************************************************************/
#include "mega16.h"
#define   uchar unsigned char
#define   uint unsigned   int
#define   CS1              PORTB.0                    //CS1 片选
#define   CS2              PORTB.1                    //CS2 片选
#define   CS3              PORTB.2                    //CS3 片选
#define   CS4              PORTB.3                    //CS4 片选
#define   CS               PORTB
#define   SE0              PORTA.0
#define   SE1              PORTA.1
#define   LED1              PORTC
#define   LED2              PORTD
uint j=0,r,q=0,t=0;
flash uchar tab1[]={                                  //显示的字符代码
      0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
       0x00,0x00,0x00,0x00,0x00,0xFF,0xFF,0xC0,
       0x00,0xC3,0x06,0x00,0x00,0xC3,0x06,0x00,
       0x00,0xC3,0x06,0x00,0x1F,0xFF,0xFF,0xF8,
```

```
0x00,0xC3,0x06,0x1C,0x00,0xC3,0x06,0x0C,
0x00,0xC3,0x06,0x0C,0x00,0xC3,0x06,0x0C,
0x00,0xFF,0xFE,0x0C,0x00,0x00,0x00,0x3C,
0x00,0x00,0x00,0x78,0x00,0x00,0x00,0x00,        /*"电",0*/
0x00,0x00,0x00,0x00,0x00,0x01,0x80,0x00,
0x00,0x01,0x80,0x00,0x0C,0x01,0x80,0x00,
0x0C,0x01,0x80,0x00,0x0C,0x01,0x80,0x08,
0x0C,0x01,0x80,0x0E,0x0C,0x3F,0xFF,0xFC,
0x0C,0x7F,0xFF,0xF8,0x0C,0xE1,0x80,0x00,
0x0D,0x81,0x80,0x00,0x0F,0x01,0x80,0x00,
0x0E,0x01,0x80,0x00,0x00,0x01,0x80,0x00,
0x00,0x01,0x80,0x00,0x00,0x00,0x00,0x00,        /*"子",1*/
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x30,
0x06,0x00,0x00,0x30,0x06,0x00,0x00,0x30,
0x06,0x00,0x00,0x30,0x06,0x00,0x00,0x30,
0x06,0x00,0x00,0x30,0x07,0xFF,0xFF,0xF0,
0x07,0xFF,0xFF,0xF0,0x06,0x00,0x00,0x30,
0x06,0x00,0x00,0x30,0x06,0x00,0x00,0x30,
0x06,0x00,0x00,0x30,0x06,0x00,0x00,0x30,
0x00,0x00,0x00,0x30,0x00,0x00,0x00,0x00,        /*"工",2*/
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x1C,
0x00,0x7F,0x00,0x1C,0x00,0x07,0xF8,0x1C,
0x00,0x00,0x18,0x1C,0x00,0x00,0x00,0x1C,
0x1F,0xFF,0xFF,0xFC,0x00,0x00,0x00,0x1C,
0x00,0x00,0x00,0x1C,0x1F,0xFF,0xFF,0xFC,
0x00,0x00,0x00,0x1C,0x00,0x00,0x18,0x1C,
0x00,0x0F,0xF0,0x1C,0x00,0x7C,0x00,0x1C,
0x00,0x00,0x00,0x1C,0x00,0x00,0x00,0x00,        /*"业",3*/
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x3F,0xF8,0x07,0xFF,0x00,0x18,
0x00,0x07,0x00,0x18,0x00,0x07,0x00,0x18,
0x00,0x07,0x00,0x18,0x3F,0xFF,0xFF,0xF8,
0x3F,0xFF,0xFF,0xF8,0x00,0x07,0x00,0x18,
0x00,0x07,0x00,0x18,0x00,0x07,0x00,0x18,
0x07,0xFF,0x3F,0xFE,0x00,0x00,0x3F,0xFE,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,        /*"出",4*/
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x3E,
0x07,0xFF,0xFF,0xF0,0x00,0x0C,0x30,0x00,
0x00,0x0C,0x30,0x00,0x1F,0xFC,0x3F,0xF8,
0x00,0x0C,0x00,0x0E,0x00,0x00,0x01,0xFC,
0x07,0xFF,0xFF,0x84,0x06,0x1E,0x00,0x0E,
0x06,0x1F,0xFE,0x3C,0x06,0x1C,0x03,0xF0,
0x0E,0x1C,0x1F,0xF8,0x0E,0x1F,0xF0,0x1C,
0x06,0x00,0x00,0x0E,0x00,0x00,0x00,0x00,        /*"版",5*/
0x00,0x00,0x00,0x00,0x00,0x60,0x1C,0x00,
0x00,0x60,0x78,0x00,0x18,0x61,0xFF,0xFE,
0x1E,0x6F,0xFF,0xFE,0x00,0x78,0x70,0x00,
0x00,0x00,0x38,0x1C,0x00,0x1C,0x00,0x1C,
```

```
        0x00,0x1C,0x00,0x1C,0x00,0x1C,0x00,0x1C,
        0x1F,0xFF,0xFF,0xFC,0x00,0x1C,0x00,0x1C,
        0x00,0x1C,0x00,0x1C,0x00,0x1C,0x00,0x1C,
        0x00,0x00,0x00,0x1C,0x00,0x00,0x00,0x00};                    /*"社",6*/
flash uchar tab2[]={0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00,          //扫描代码
                    0x0F,0x0E,0x0D,0x0C,0x0B,0x0A,0x09,0x08};
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<114;n++)
       {;}
    }
}
void display(void)
{
  for(r=0;r<30;r++)                                                  //控制每一个字符显示的时间
       for(j=q;j<64+q;j++)
        {
            SE0=0;
            SE1=1;
            CS=tab2[t];
       LED1=tab1[j];
       j++;
       LED2=tab1[j];
            delay(10);
            j++;
      SE0=1;
            SE1=0;
            CS=tab2[t];
       LED1=tab1[j];
       j++;
       LED2=tab1[j];
            delay(10);
            t++;
       if(t= =16)
          {
                  t=0;
            }
          }
      q=q+64;                                                        //显示下一个字符
          if(q= =448)
           {
              q=0;
           }
}
void port_init(void)                                                 //端口初始化
{
```

```
    DDRA=0xFF;
   PORTA=0xFF;
    DDRB=0xFF;
   PORTB=0xFF;
    DDRC=0xFF;
   PORTC=0xFF;
    DDRD=0xFF;
   PORTD=0xFF;
}
void main(void)
{
    port_init();
    while(1)
      {
        display();
      }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“两个 16×16LED 点阵汉字显示.COFF”文件，然后在 Proteus ISIS 中打开已创建的两个 16×16LED 点阵汉字显示电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试运行状态下，可看见两个 16×16LED 点阵汉字显示效果，如图 9-32 所示。

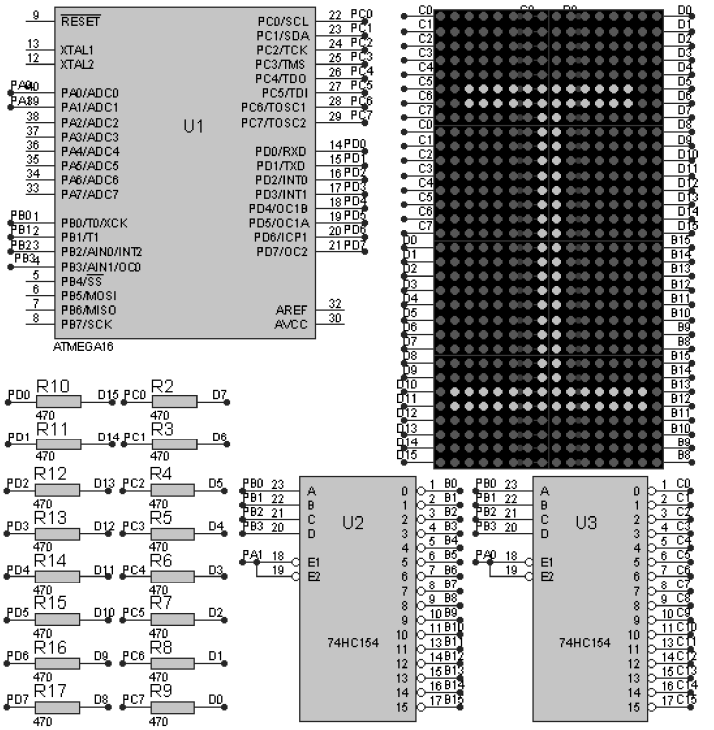


图 9-32　两个 16×16LED 点阵汉字显示仿真效果图

任务 9　两个 16×16 LED 点阵汉字分批显示

使用两个 16×16 共阴极 LED 点阵分批显示字符串“基于 Proteus 的 ATmega16 单片机程序开发”。

硬件设计

本任务的硬件电路可采用任务 8 中图 9-30 所示的原理图。

软件设计

由于该点阵每一屏需显示 2 个字，要显示“基于 Proteus 的 C51 单片机程序开发”字符串，应该分 8 屏显示，显示次序为“基于”→“Prot”→“eus()”→“的 AT”→“mega”→“16 单”→“片机”→“程序”→“开发”，“()”表示半个汉字的空白。由于在此增加了一些空白显示，因此该字符串库中有 576 个段码值。其程序流程图如图 9-33 所示。

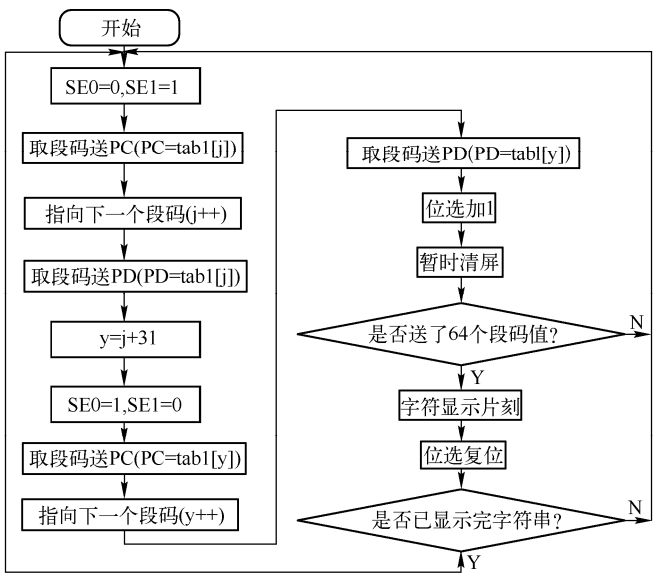


图 9-33　两个 16×16 LED 点阵汉字分批显示程序流程图

源程序

```
/************************************************************
  文件名:          两个 16×16LED 点阵汉字移位显示.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
  说明:            在 16×16LED 点阵上显示　基于 Proteus 的 ATmega16 单片机程序开发
  取模方法:        Times NEew Roman、共阴极、取模式方式为逐列式、高位在前
************************************************************/
#include "mega16.h"
```

```
#define   uchar unsigned char
#define   uint unsigned   int
#define   CS1              PORTB.0                    //CS1 片选
#define   CS2              PORTB.1                    //CS2 片选
#define   CS3              PORTB.2                    //CS3 片选
#define   CS4              PORTB.3                    //CS4 片选
#define   CS               PORTB
#define   SE0              PORTA.0
#define   SE1              PORTA.1
#define   LED1             PORTC
#define   LED2             PORTD
uint j=0,r,q=0,t=0,y=0;
flash uchar tab1[]={                                  //显示的字符代码
        0x00,0x88,0x20,0x88,0x20,0x91,0x20,0xA1,
        0xFF,0xC9,0x2A,0x89,0x2A,0x89,0x2A,0xBF,
        0x2A,0x89,0x2A,0x89,0xFF,0xC9,0x20,0xA1,
        0x20,0x91,0x20,0x88,0x00,0x88,0x00,0x00,      /*"基",0*/
        0x02,0x00,0x02,0x00,0x42,0x00,0x42,0x00,
        0x42,0x00,0x42,0x02,0x42,0x01,0x7F,0xFE,
        0x42,0x00,0x42,0x00,0x42,0x00,0x42,0x00,
        0x42,0x00,0x02,0x00,0x02,0x00,0x00,0x00,      /*"于",1*/
        0x10,0x08,0x10,0x08,0x1F,0xF8,0x1F,0xF8,
        0x10,0x88,0x10,0x88,0x10,0x80,0x19,0x80,      /*"P",2-1*/
        0x01,0x08,0x03,0xF8,0x03,0xF8,0x01,0x08,
        0x02,0x00,0x02,0x00,0x00,0x00,0x00,0x00,      /*"r",2-2*/
        0x00,0x00,0x00,0xE0,0x01,0xF0,0x03,0x18,
        0x02,0x08,0x02,0x08,0x03,0x08,0x01,0xF0,      /*"o",3-1*/
        0x02,0x00,0x07,0xF8,0x0F,0xF8,0x02,0x08,
        0x02,0x10,0x00,0x00,0x00,0x00,0x00,0x00,      /*"t",3-2*/
        0x00,0x00,0x00,0xE0,0x01,0xF0,0x02,0x98,
        0x02,0x88,0x02,0x88,0x03,0x88,0x01,0x90,      /*"e",4-1*/
        0x02,0x00,0x03,0xF0,0x03,0xF8,0x00,0x08,
        0x00,0x08,0x02,0x10,0x03,0xF8,0x03,0xF8,      /*"u",4-2*/
        0x00,0x00,0x01,0x98,0x02,0xC8,0x02,0x48,
        0x02,0x48,0x01,0x68,0x03,0x30,0x00,0x00,      /*"s",5-1*/
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,      /*" ",5-2*/
        0x00,0x00,0x1F,0xFE,0x30,0x84,0xD0,0x84,
        0x10,0x84,0x10,0x84,0x1F,0xFE,0x02,0x00,
        0x0C,0x00,0xF1,0x00,0x10,0xC2,0x10,0x01,
        0x10,0x02,0x1F,0xFC,0x00,0x00,0x00,0x00,      /*"的",6*/
        0x00,0x08,0x00,0x08,0x00,0x38,0x00,0x48,
        0x01,0xC0,0x06,0x40,0x18,0x40,0x1E,0x40,      /*"A",7-1*/
        0x18,0x00,0x10,0x00,0x10,0x00,0x10,0x08,
        0x10,0x08,0x1F,0xF8,0x1F,0xF8,0x10,0x08,      /*"T",7-2*/
        0x01,0x08,0x03,0xF8,0x03,0xF8,0x01,0x08,
        0x02,0x00,0x02,0x08,0x03,0xF8,0x01,0xF8,      /*"m",8-1*/
        0x00,0x00,0x00,0xE0,0x01,0xF0,0x02,0x98,
```

```
        0x02,0x88,0x02,0x88,0x03,0x88,0x01,0x90,                    /*"e",8-2*/
        0x00,0x02,0x01,0x9D,0x03,0xE9,0x02,0x29,
        0x02,0x29,0x02,0x29,0x03,0xE9,0x03,0xCA,                    /*"g",9-1*/
        0x00,0x00,0x01,0x18,0x03,0x38,0x02,0x48,
        0x02,0x48,0x03,0xF0,0x01,0xF8,0x00,0x08,                    /*"a",9-2*/
        0x00,0x00,0x08,0x08,0x08,0x08,0x1F,0xF8,
        0x1F,0xF8,0x00,0x08,0x00,0x08,0x00,0x00,                    /*"1",10-1*/
        0x01,0xE0,0x03,0xF0,0x05,0x18,0x0A,0x08,
        0x0A,0x08,0x13,0x18,0x13,0xF0,0x11,0xE0,                    /*"6",10-2*/
        0x00,0x08,0x00,0x08,0x1F,0xC8,0x92,0x48,
        0x52,0x48,0x32,0x48,0x12,0x48,0x1F,0xFF,
        0x12,0x48,0x32,0x48,0x52,0x48,0x92,0x48,
        0x1F,0xC8,0x00,0x08,0x00,0x08,0x00,0x00,                    /*"单",11*/
        0x00,0x00,0x00,0x01,0x00,0x06,0x7F,0xF8,
        0x04,0x40,0x04,0x40,0x04,0x40,0x04,0x40,
        0x04,0x40,0xFC,0x40,0x04,0x7F,0x04,0x00,
        0x04,0x00,0x04,0x00,0x00,0x00,0x00,0x00,                    /*"片",12*/
        0x08,0x20,0x08,0xC0,0x0B,0x00,0xFF,0xFF,
        0x09,0x00,0x08,0xC1,0x00,0x06,0x7F,0xF8,
        0x40,0x00,0x40,0x00,0x40,0x00,0x7F,0xFC,
        0x00,0x02,0x00,0x02,0x00,0x1E,0x00,0x00,                    /*"机",13*/
        0x24,0x10,0x24,0x60,0x25,0x80,0x7F,0xFF,
        0xC4,0x80,0x44,0x60,0x00,0x02,0x7C,0x92,
        0x44,0x92,0x44,0x92,0x44,0xFE,0x44,0x92,
        0x44,0x92,0x7C,0x92,0x00,0x82,0x00,0x00,                    /*"程",14*/
        0x00,0x02,0x00,0x0C,0x3F,0xF0,0x20,0x00,
        0x20,0x80,0x20,0x80,0x28,0x80,0xA8,0x82,
        0x6A,0x81,0x29,0xFE,0x2A,0x80,0x2C,0x80,
        0x28,0x80,0x20,0xA0,0x20,0xC0,0x00,0x00,                    /*"序",15*/
        0x01,0x00,0x41,0x01,0x41,0x02,0x41,0x0C,
        0x7F,0xF0,0x41,0x00,0x41,0x00,0x41,0x00,
        0x41,0x00,0x41,0x00,0x7F,0xFF,0x41,0x00,
        0x41,0x00,0x41,0x00,0x01,0x00,0x00,0x00,                    /*"开",16*/
        0x00,0x00,0x00,0x04,0x18,0x08,0x68,0x31,
        0x08,0xC1,0x0B,0x01,0x1D,0x82,0xE9,0x62,
        0x09,0x14,0x09,0x08,0x09,0x14,0x49,0x22,
        0x29,0xC2,0x08,0x01,0x00,0x01,0x00,0x00};                   /*"发",17*/
flash uchar tab2[]={0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00,        //扫描代码
                    0x0F,0x0E,0x0D,0x0C,0x0B,0x0A,0x09,0x08};
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<114;n++)
        {;}
    }
}
```

```
void display(void)
{
  for(r=0;r<30;r++)                                    //控制每一个字符显示的时间
    for(j=q;j<32+q;j++)
      {
          SE0=0;
          SE1=1;
          CS=tab2[t];
          LED1=tab1[j];
          j++;
          LED2=tab1[j];
          delay(10);                                   //上半部的点阵显示字
          y=j+31;
          SE0=1;
          SE1=0;
          CS=tab2[t];
          LED1=tab1[y];
          y++;
          LED2=tab1[y];
          delay(10);
          t++;
            if(t= =16)
                  {
                  t=0;
              }
       }
      q=q+64;                                          //显示下一个字符
      if(q= =576)
          {
              q=0;
          }
}
void port_init(void)                                   //端口初始化
{
    DDRA=0xFF;
   PORTA=0xFF;
    DDRB=0xFF;
   PORTB=0xFF;
    DDRC=0xFF;
   PORTC=0xFF;
    DDRD=0xFF;
   PORTD=0xFF;
}
void main(void)
{
   port_init();
   while(1)
      {
```

```
            display();
        }
    }
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“两个 16×16LED 点阵汉字分批显示.COFF”文件，然后在 Proteus ISIS 中打开已创建的两个 16×16LED 点阵汉字分批显示电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试运行状态下，可看见两个 16×16LED 点阵汉字分批显示效果，如图 9-34 所示。

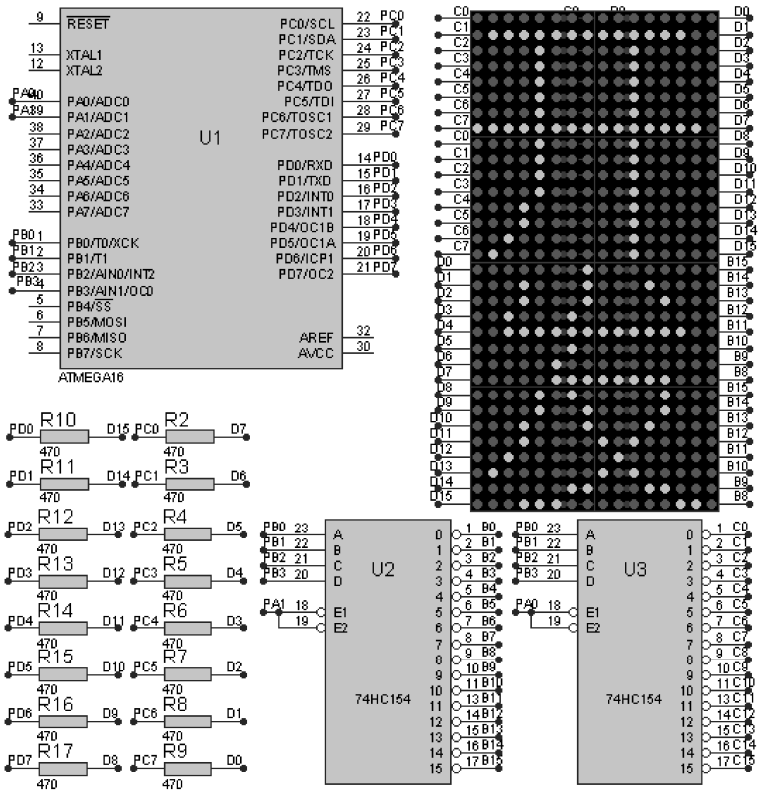


图 9-34　两个 16×16 LED 点阵汉字分批显示仿真效果图

任务 10　两个 16×16 LED 点阵汉字移位显示

使用两个 16×16 共阴极 LED 点阵移位显示字符串“湖南省长沙市岳麓山”。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 9-4 所列的元器件，并完成图 9-35 所示的硬件电路图设计。

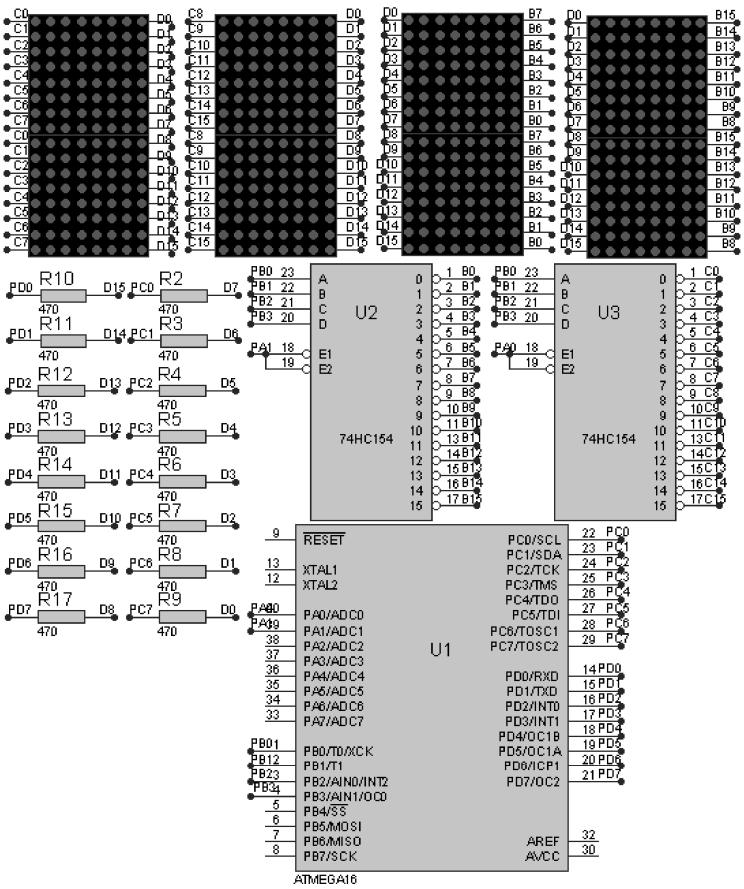


图 9-35　两个 16×16 LED 点阵汉字移位显示电路图

> 注意：
> 图中单片机的晶振电路、复位电路等部分没有绘制出。

软件设计

两个 16×16 LED 点阵汉字移位显示是在两个 16×16 LED 点阵汉字显示的基础上进行的，将两个 16×16 LED 点阵汉字显示的源程序稍加修改即可。为了更好观看到移位显示效果，需要“湖南”前添加 2 个字的空白字符，即 64 个 0x00 代码；在“山”字后添加 1 个字的空白字符，即 32 个 0x00 代码。

源程序

```
/************************************************************
  文件名:          两个 16×16LED 点阵汉字移位显示.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
  说明:            在 16×16LED 点阵上显示　湖南省长沙市岳麓山
  取模方法:        楷体_GB2312、共阴极、取模式方式为逐列式、高位在前
************************************************************/
#include "mega16.h"
```

```
#define   uchar unsigned char
#define   uint unsigned   int
#define   CS1              PORTB.0                    //CS1 片选
#define   CS2              PORTB.1                    //CS2 片选
#define   CS3              PORTB.2                    //CS3 片选
#define   CS4              PORTB.3                    //CS4 片选
#define   CS               PORTB
#define   SE0               PORTA.0
#define   SE1               PORTA.1
#define   LED1              PORTC
#define   LED2              PORTD
uint j=0,r,q=0,t=0,y=0;
flash uchar tab1[]={                                  //显示的字符代码
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,      //什么都不显示
        0x00,0x00,0x04,0x18,0x22,0xE0,0x10,0x00,
        0x04,0xE0,0x05,0x20,0xFF,0x20,0x09,0xC4,
        0x08,0x08,0x00,0x30,0x1F,0xC0,0x24,0x80,
        0x20,0x04,0x3F,0xFC,0x00,0x00,0x00,0x00,      /*"湖",1*/
        0x00,0x00,0x00,0x00,0x01,0xF8,0x02,0x00,
        0x12,0x50,0x13,0x50,0x1E,0xD0,0xF2,0x78,
        0x23,0xA0,0x24,0xA0,0x24,0x08,0x04,0x04,
        0x07,0xFC,0x00,0x00,0x00,0x00,0x00,0x00,      /*"南",2*/
        0x00,0x10,0x00,0x10,0x00,0x20,0x04,0x40,
        0x18,0x40,0x00,0x80,0x01,0xFC,0xFA,0xA8,
        0x06,0xA8,0x1A,0x08,0x02,0x04,0x43,0xFC,
        0x20,0x00,0x00,0x00,0x00,0x00,0x00,0x00,      /*"省",3*/
        0x00,0x00,0x01,0x00,0x01,0x00,0x01,0x00,
        0x01,0x00,0x7F,0xFC,0x05,0x08,0x0A,0x90,
        0x12,0x40,0x62,0x20,0x02,0x10,0x02,0x18,
        0x02,0x08,0x00,0x08,0x00,0x00,0x00,0x00,      /*"长",4*/
        0x00,0x00,0x00,0x00,0x04,0x30,0x42,0xC0,
        0x20,0x02,0x01,0x02,0x06,0x04,0x00,0x04,
        0x00,0x08,0xFF,0x10,0x00,0x20,0x00,0xC0,
        0x13,0x00,0x10,0x00,0x08,0x00,0x00,0x00,      /*"沙",5*/
        0x00,0x00,0x08,0x00,0x08,0x00,0x08,0x00,
        0x0B,0xE0,0x0A,0x00,0x8A,0x00,0x4F,0xFE,
        0x14,0x00,0x14,0x40,0x14,0x20,0x17,0xC0,
        0x10,0x00,0x10,0x00,0x00,0x00,0x00,0x00,      /*"市",6*/
        0x00,0x80,0x00,0x80,0x00,0x80,0x00,0x88,
        0x10,0xB8,0x1F,0x88,0x25,0x08,0x25,0x78,
        0x4F,0x10,0x49,0x10,0x09,0x10,0x09,0x7C,
```

```
        0x01,0x00,0x01,0x00,0x01,0x00,0x00,0x00,                    /*"岳",7*/
        0x00,0x02,0x00,0x04,0x14,0x18,0x19,0xE0,
        0x12,0x22,0xFE,0xBE,0x2B,0xD4,0x15,0x40,
        0x27,0xFC,0xFD,0x4A,0x65,0xD2,0x54,0x32,
        0x48,0x02,0x08,0x0E,0x08,0x00,0x00,0x00,                    /*"麓",8*/
        0x00,0x00,0x00,0x00,0x01,0xF8,0x00,0x08,
        0x00,0x10,0x00,0x10,0x00,0x10,0x7F,0xE0,
        0x00,0x20,0x00,0x20,0x00,0x20,0x00,0x20,
        0x03,0xF8,0x00,0x00,0x00,0x00,0x00,0x00,                    /*"山",9*/
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};              //什么都不显示
flash uchar tab2[]={0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00,        //扫描代码
                    0x0F,0x0E,0x0D,0x0C,0x0B,0x0A,0x09,0x08};
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<114;n++)
        {;}
    }
}
void display(void)
{
  for(r=0;r<40;r++)                                                 //控制每一个字符显示的时间
    for(j=q;j<32+q;j++)
      {
          SE0=0;
          SE1=1;
          CS=tab2[t];
          LED1=tab1[j];
          j++;
          LED2=tab1[j];
          delay(5);
          y=j+31;
          SE0=1;
          SE1=0;
          CS=tab2[t];
          LED1=tab1[y];
          y++;
          LED2=tab1[y];
          delay(5);
          t++;
           if(t= =16)
                  {
                  t=0;
```

```
            }
        }
        q=q+16;                                          //显示下一个字符
        if(q= =352)
            {
                q=0;
            }
}
void port_init(void)                                     //端口初始化
{
     DDRA=0xFF;
    PORTA=0xFF;
     DDRB=0xFF;
    PORTB=0xFF;
     DDRC=0xFF;
    PORTC=0xFF;
     DDRD=0xFF;
    PORTD=0xFF;
}
void main(void)
{
    port_init();
    while(1)
      {
        display();
      }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“两个 16×16LED 点阵汉字移位显示.COFF”文件，然后在 Proteus ISIS 中打开已创建的两个 16×16LED 点阵汉字移位显示电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试运行状态下，可看见两个 16×16LED 点阵汉字分批显示效果，如图 9-36 所示。

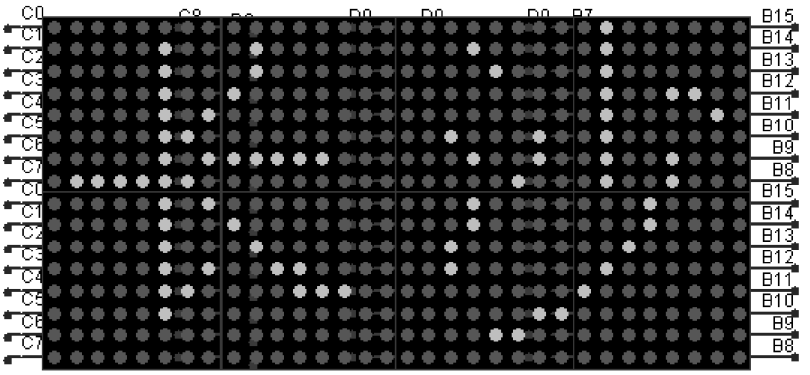


图 9-36　两个 16×16 LED 点阵汉字移位显示仿真效果图

项目十　LCD 显示设计

〖知识目标〗
☺ 了解液晶显示器（LCD）的结构及工作原理;
☺ 理解字符式 LCD 的控制。
〖能力目标〗
☺ 掌握字符式 LCD 显示字符的原理;
☺ 能够熟练使用字符式 LCD 进行字符显示。

液晶显示器（Liquid Crystal Display，LCD）是一种利用液晶的扭曲/向列效应制成的新型显示器。它具有体积小、质量轻、功耗低、抗干扰能力强等优点，因而在单片机系统中被广泛应用。

任务 1　LCD 显示原理认知

1．LCD 的结构及工作原理

LCD 本身不发光，是通过借助外界光线照射液晶材料而实现显示的被动显示器件。LCD 的基本结构如图 10-1 所示。

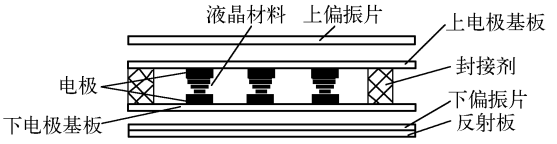


图 10-1　LCD 的基本结构

向列型液晶材料被封装在上（正）、下（背）两片导电玻璃电极之间。液晶分子垂直排列，上、下扭曲 90°。外部入射光线通过上偏振片后形成偏振光，该偏振光通过平行排列的液晶材料后被旋转 90°，再通过与上偏振片垂直的下偏振片，被反射板反射过来，呈透明状态。若在其上、下电极上加上一定的电压，在电场的作用下迫使加在电极部分的液晶分子转成垂直排列，其旋光作用也随之消失，致使从上偏振片入射的偏振光不被旋转，光无法通过下偏振片返回，呈黑色。当去掉电压后，液晶分子又恢复其扭转结构。因此可以根据需要将电极做成各种形状，用以显示各种文字、数字、图形。

2．LCD 分类

LCD 分类的方法有多种。

【按电光效应分类】电光效应是指在电的作用下，液晶分子的初始排列改变为其他的排列形式，使液晶盒的光学性质发生变化，即以电通过液晶分子对光进行了调制。

LCD 按电光效应的不同，可分为电场效应类、电流效应类、电热效应类 3 种。电场效应类又可分为扭曲向列效应（Twisted Nematic，TN）型、宾主效应（GH）型和超扭曲效应（Super Twisted，STN）型等。

目前在单片机应用系统中广泛应用 TN 型和 STN 型 LCD。

【按显示内容分类】LCD 按其显示的内容不同，可分为字段式（又称笔画式）、点阵字符式和点阵图 3 种。

☺ 字段式 LCD 是以长条笔画状显示像素组成的 LCD。

☺ 点阵字符式有 192 种内置字符，包括数字、字母、常用标点符号等。另外，用户可以自定义 5×7 点阵字符或其他点阵字符等。根据 LCD 型号的不同，每屏显示的行数有 1 行、2 行、4 行 3 种，每行可显示 8 个、16 个、20 个 24 个、32 个和 40 个字符等。

☺ 点阵图形式的 LCD 液晶显示器除可以显示字符外，还可显示各种图形信息、汉字等。

【按采光方式分类】LCD 按采光方式的不同，可分为带背光源和不带背光源两类。

不带背光源 LCD 是靠显示器背面的反射膜将射入的自然光从下面反射出来完成的。大部分设备的 LCD 是用自然光的光源，可选用不带背光的 LCD 器件。

若产品工作在弱光或黑暗条件下时，就应选择带背光的 LCD 显示器。

3．LCD 的驱动方式

LCD 两极间不允许施加恒定直流电压，驱动电压的直流成分越小越好，最好不超过 50mV。为了得到 LCD 亮、灭所需的两倍幅值及零电压，常给 LCD 的背极通以固定的交变电压，通过控制前极电压值的改变实现对 LCD 显示的控制。

LCD 的驱动方式由电极引线的选择方式确定。其驱动方式有静态驱动（直接驱动）和时分割驱动（也称多极驱动或动态驱动）两种。

【静态驱动方式】静态驱动是把所有段电极逐个驱动，所有段电极和公共电极之间仅在要显示时才施加电压。静态驱动是液晶显示器最基本的驱动方式，其驱动原理电路及波形如图 10-2 所示。

图中 LCD 表示某个液晶显示字段。字段波形 C 与公用波形 B 不是同相就是反相。当此字段上两个电极电压相位相同时，两个电极的相对电压为零，液晶上无电场，该字段不显示；当此字段上两个电极的电压相位相反时，两个电极的相对电压为两倍幅值方波电压，该字段呈黑色显示。

在静态驱动方式下，若 LCD 有 n 个字段，则需 n+1 条引线，其驱动电路也需要 n+1 条引线。当显示字段较多时，驱动电路的引线数将需更多。所以当显示字段较少时，一般采用静态驱动方式。当显示字段较多时，一般采用时分割驱动方式。

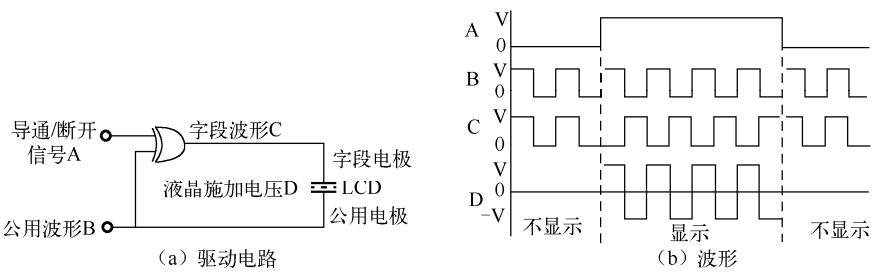


图 10-2　LCD 静态驱动原理电路及波形

【时分割驱动方式】时分割驱动是把全段电极集分为数组，将它们分时驱动，即采用逐

行扫描的方法显示所需要的内容。时分割驱动原理如图 10-3 所示。

从图 10-3 中可以看出，电极沿 X、Y 方向排列成矩阵形式，按顺序给 X 电极施加选通波形，给 Y 电极施加与 X 电极同步的选通或非选通波形，如此周而复始。在 X 电极与 Y 电极交叉的段点被点亮或熄灭，达到 LCD 显示的目的。

驱动 X 电极从第一行到最后一行所需时间为帧周期 T_f，驱动每一行所需时间 T_r 与帧周期 T_f 的比值为占空比 Duty。

时分割的占空为：Duty=T_r/T_f=1/n。其占空比有 1/2、1/8、1/11、1/16、1/32、1/64 等。非选通时波形电压与选通时波形电压的比值称为偏比 Bias，Bias=1/a（a 为 Duty 的平方根加 1）。其偏比有 1/2、1/3、1/4、1/5、1/7、1/9 等。

图 10-4 所示为一位 8 段 1/3 偏比的 LCD 数码管各字段与背极的排列、等效电路。

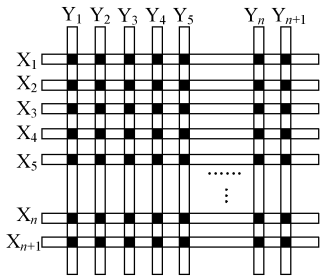


图 10-3　LCD 时分割驱动原理

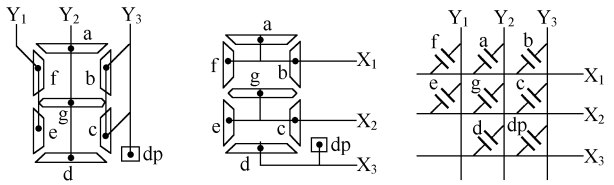


图 10-4　一位 LCD 数码管各字段与背极的排列、等效电路图

从图 10-4 中可以看出，3 根公共电极 X_1、X_2、X_3 分别与所有字符的 a、b、f；c、e、g；d、dp 相连，而 Y_1、Y_2、Y_3 是每个字符的单独电极，分别与 f、e；a、d、g；b、c、dp 相连。通过这种分组的方法可使具有 m 个字符段的 LCD 的引脚数为$\frac{m}{n}+n$（n 为背极数），减少了驱动电路的引线数。所以当显示像素众多时，如点阵型 LCD，为节省驱动电路，多采用时分割驱动方式。

4．SMC1602A 字符式 LCD 的基础知识

SMC1602A 显示器可以显示两行字符，每行 16 个，显示容量为 16×2 字符。它带有背光源，采用时分割驱动的形式。通过并行接口，可与单片机 I/O 口直接相连。

【SMC1602A 的引脚及其功能】SMC1602A 采用并行接口方式，有 16 根引线，各线的功能及使用方法如下所述。

☺ V_{SS}（1）：电源地。

☺ V_{DD}（2）：电源正极，接+5V 电源。

☺ V_L（3）：液晶显示偏压信号。

☺ RS（4）：数据/指令寄存器选择端。高电平时选择数据寄存器，低电平时选择指令寄存器。

☺ R/W（5）：读/写选择端。高电平时为读操作，低电平时为写操作。

☺ E（6）：使能信号，下降沿触发。

☺ D_0 ~ D_7（7 ~ 14）：I/O 数据传输线。

☺ BLA（15）：背光源正极。

☺ BLK（16）：背光源负极。

【SMC1602A 内部结构及工作原理】SMC1602A LCD 内部主要由日立公司的 HD44780、HD44100（或兼容电路）和数个电阻、电容等部分组成。

HD44780 是用低功耗 CMOS 技术制造的大规模点阵 LCD 控制器，具有简单而功能较强的指令集，可实现字符移动、闪烁等功能，与微处理相连能使 LCD 显示大小英文字母、数字和符号。HD44780 控制电路主要由 DDRAM、CGROM、CGRAM、IR、DR、BF、AC 等大规模集成电路组成。

☺ DDRAM（Data Display RAM）为数据显示用的 RAM，用以存放要 LCD 显示的数据，能存储 80 个，只要将标准的 ASCII 码放入 DDRAM，内部控制线路就会自动将数据传送到显示器上，并显示出该 ASCII 码对应的字符。

☺ CGROM（Character Generator ROM）为字符产生器 ROM，它存储了由 8 位字符码生成的 192 个 5×7 点阵字型和 32 种 5×10 点阵字符，8 位字符编码和字符的对应关系（即内置字符集）见表 10-1。

表 10-1 HD44780 内置字符集

高4位 / 低4位	0000	0001	0010	0011	0100	0101	0110	0111	1010	1011	1100	1101	1110	1111
xxxx0000	CGRA			0	@	P	‘	p		ー	タ	ミ	α	P
xxxx0001	(2)		!	1	A	Q	a	q	。	ア	チ	ム	ä	q
xxxx0010	(3)		"	2	B	R	b	r	「	イ	ツ	メ	β	θ
xxxx0011	(4)		#	3	C	S	c	s	」	ウ	テ	モ	ε	∞
xxxx0100	(5)		$	4	D	T	d	t	、	エ	ト	ヤ	μ	Ω
xxxx0101	(6)		%	5	E	U	e	u	・	オ	ナ	ユ	B	0
xxxx0110	(7)		&	6	F	V	f	v	ヲ	カ	ニ	ヨ	ρ	Σ
xxxx0111	(8)		’	7	G	W	g	w	ア	キ	ヌ	ラ	9	π
xxxx1000	(1)		(	8	H	X	h	x	イ	ク	ネ	リ	ſ	x̄
xxxx1001	(2)		)	9	I	Y	i	y	ウ	ケ	ノ	ル	-¹	4
xxxx1010	(3)		*	:	J	Z	j	z	エ	コ	ハ	レ	j	÷
xxxx1011	(4)		+	;	K	[	k	(	オ	サ	ヒ	ロ	ˣ	J
xxxx1100	(5)		,	<	L	¥	l	\|	ヤ	シ	フ	ワ	¢	円
xxxx1101	(6)		-	=	M	]	m	)	ユ	ス	ヘ	ン	Ł	÷
xxxx1110	(7)		.	>	N	^	n	→	ヨ	セ	ホ	゜	ñ	
xxxx1111	(8)		/	?	O	_	o	←	ッ	ソ	マ	゜	ö	

☺ CGRAM（Character Generator RAM）为字型、字符产生器，可供使用者存储特殊造型的字型码，CGRAM 最多可存 8 个字型。

☺ IR（Instruction Register）为指令寄存器，负责存储 MCU 要写给 LCD 的指令码，当 RS 及 R/W 引脚信号为 0 且 E[Enable]引脚信号由 1 变为 0 时，D0～D7 引脚上的数据便会存入到 IR 寄存器中。

☺ DR（Data Register）为数据寄存器，它们负责存储 MCU 要写到 CGRAM 或 DDRAM 的数据，或者存储 MCU 要从 CGRAM 或 DDRAM 读出的数据。因此，可将 DR 视为一个数据缓冲区，当 RS 及 R/W 引脚信号为 1 且 E[Enable]引脚信号由 1 变为 0 时，读取数据；当 RS 引脚信号为 1，R/W 引脚信号为 0 且 E[Enable]引脚信号由 1 变为 0 时，存入数据。

☺ BF（Busy Flag）为忙碌信号，当 BF 为 1 时，不接收 MCU 送来的数据或指令；当 BR 为 0 时，接收外部数据或指令。所以，在写数据或指令到 LCD 前，必须查看 BF 是否为 0。

☺ AC（Address Counter）为地址计数器，负责计数写入/读出 CGRAM 或 DDRAM 的数据地址，AC 依照 MCU 对 LCD 的设置值而自动修改它本身的内容。

HD44100 也是采用 CMOS 技术制造的大规模 LCD 驱动 IC，既可当行驱动用，又可当列驱动用，由 20×2Bit 二进制移位寄存器、20×2Bit 数据锁存器、20×2Bit 驱动器组成，主要用于 LCD 时分割驱动。

【显示位与 RAM 的对应关系（地址映射）】SMC1602A 内部带有 80×8Bit 的 RAM 缓冲区，显示位与 RAM 的对应关系见表 10-2。

表 10-2　显示位与 RAM 地址的对应关系

显示位序号		1	2	3	4	5	6	……	40
RAM 地址(HEX)	第 1 行	00	01	02	03	04	05	……	27
	第 2 行	40	41	42	43	44	06	……	67

【指令操作】指令操作包括清屏、回车、输入模式控制、显示开关控制、移位控制、显示模式控制等，见表 10-3。

表 10-3　指令系统

指令名称	控制信号		指令代码								功　能
	RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0	
清屏	0	0	0	0	0	0	0	0	0	1	显示清屏：（1）数据指针清零，（2）所有显示清除
回车	0	0	0	0	0	0	0	0	1	0	显示回车，数据指针清零
输入模式控制	0	0	0	0	0	0	0	1	N	S	设置光标、显示画面移动方向
显示开关控制	0	0	0	0	0	0	D/L	D	C	B	设置显示、光标、闪烁开关
移位控制	0	0	0	0	0	1	S/C	R/L	×	×	使光标或显示画面移位
显示模式控制	0	0	0	0	1	D/L	N	F	×	×	设置数据总线位数、点阵方式
CGRAM 地址设置	0	0	0	1	ACG						
DDRAM 地址指针设置	0	0	1	ADD							
忙状态检查	0	1	BF	AC							
读数据	1	1	数　据								从 RAM 中读取数据
写数据	1	0	数　据								对 RAM 进行写数据
数据指针设置	0	0	80H+地址码（0～27H，40～47H）								设置数据地址指针

〖说明〗表中的“×”表示“0”或“1”，下同。

☺ 清屏指令：设置清屏指令，使 DDRAM 的显示内容清零、数据指针 AC 清零，光标回到左上角的原点。

☺ 回车指令：设置回车指令，显示回车，数据指针 AC 清零，使光标和光标所在的字符回到原点，便 DDRAM 单元的内容不变。

☺ 输入模式控制指令：输入模式控制指令，用于设置光标、显示画面移动方向。当数据写入 DDRAM（CGRAM）或从 DDRAM（CGRAM）读取数据时，N 控制 AC 自动加 1 或自动减 1。若 N 为 1 时，AC 加 1；N 为 0 时，AC 减 1。S 控制显示内容左移或右移，S=1 且数据写入 DDRAM 时，显示将全部左移（N=1）或右移（N=0），此时光标看上去未动，仅显示内容移动，但读出时显示内容不移动；当 S=0 时，显示不移动，光标左移或右移。

☺ 显示开关控制指令：显示开关控制指令，用于设置显示、光标、闪烁开关。D 为显示控制位，当 D=1 时，开显示；当 D=0 时，关显示，此时 DDRAM 的内容保持不变。C 为光标控制位，当 C=1 时，开光标显示；C=0 时，关光标显示。B 为闪烁控制位，当 B=1 时，当光标和光标所指的字符共同以 1.25Hz 速率闪烁；B=0 时，不闪烁。

☺ 移位控制指令：移位控制指令，使光标或显示画面在没有对 DDRAM 进行读/写操作时被左移或右移。该指令每执行 1 次，屏蔽字符与光标即移动 1 次。在两行显示方式下，光标为闪烁的位置从第 1 行移到第 2 行。移位控制指令的设置见表 10-4。

表 10-4　移位控制指令的设置

D7～D4	D3	D2	D1	D0	指令设置含义
	S/C	R/L			
0001	0	0	×	×	光标左移，AC 自动减 1
0001	0	1	×	×	光标移位，光标和显示一起右移
0001	1	0	×	×	显示移位，光标左移，AC 自动加 1
0001	1	1	×	×	光标和显示一起右移

☺ 显示模式控制指令：显示模式控制指令，用于设置数据总线位数、点阵方式等操作，见表 10-5。

表 10-5　显示模式控制指令的设置

D7～D5	D4	D3	D2	D1	D0	指令设置含义
	D/L	N	F			
001	1	1	1	×	×	DL=1 选择 8 位数据总线；N=1 两行显示；F=1 为 5×10 点阵
001	1	1	0	×	×	DL=1 选择 8 位数据总线；N=1 两行显示；F=0 为 5×7 点阵
001	1	0	1	×	×	DL=1 选择 8 位数据总线；N=0 一行显示；F=1 为 5×10 点阵
001	1	0	0	×	×	DL=1 选择 8 位数据总线；N=0 一行显示；F=0 为 5×7 点阵
001	0	1	1	×	×	DL=0 选择 4 位数据总线；N=1 两行显示；F=1 为 5×10 点阵
001	0	0	1	×	×	DL=0 选择 4 位数据总线；N=0 一行显示；F=1 为 5×10 点阵
001	0	0	0	×	×	DL=0 选择 4 位数据总线；N=0 一行显示；F=0 为 5×7 点阵

☺ CGRAM 地址设置指令：CGRAM 地址设置指令，用于设置 CGRAM 地址指针，地址码 D5～D7 被送入 AC。设置此指令后，就可以将用户自己定义的显示字符数据写入 CGRAM 或从 CGRAM 中读出。

☺ DDRAM 地址指针设置指令：DDRAM 地址指针设置指令用于设置两行字符显示的起始地址。为 10000000（0x80H）时，设置第 1 行字符的显示位置为第 1 行第 0 列，为 0x81～0x8F 时，为第 1 行第 1 列～第 1 行第 15 列。为 11000000（0xC0H）时，设置第 2 行字符的显示位置为第 2 行第 0 列，为 0xC1～0xCF 时，为第 2 行第 2 列～第 2 行第 15 列。此指令设置 DDRAM 地址指针的值，此后就可以将要显示的数据写入到 DDRAM 中。在 HD44780 控制器中，由于内嵌大量的常用字符，这些字符都集成在 CGROM 中，当要显示这些点阵时，只需将该字符所对应的字符代码送给指定的 DDRAM 中即可。

☺ 忙状态检查指令：忙状态检查指令是通过读取数据的 D7 位是否为 1，若为 1 表示总线正在忙碌。

【CVAVR 中 LCD 功能函数】LCD 功能函数针对由日立 HD44780 或兼容芯片控制的字符型 LCD 模块，在 CVAVR 中，与 LCD 字符显示器有关的 LCD 功能函数主要有如下 7 种。

☺ void lcd_init(unsigned char lcd_columns)：该函数对 LCD 进行初始化，并清除 LCD 的显示，将显示位置回到第 0 行的第 0 列的起始位置处。函数的参数应是 LCD 显示器的列数（一行能够显示的字符数）。使用 LCD 时，必须先使用该函数对 LCD 进行初始化，然后才能使用其他高级的 LCD 功能函数。

☺ void lcd_clear(void)：该函数清除 LCD 的显示，并将显示位置回到第 0 行的第 0 列的起始位置处。

☺ void lcd_gotoxy(unsigned char x, unsigned char y)：该函数将显示位置定位于第 x 列的第 y 行的位置处。

> 注意：
> LCD 的行列定位都是从“0”起始的。

☺ void lcd_putchar(char c)：该函数将字符 c 在当前的显示位置上显示出来。

☺ void lcd_puts(char *str)：该函数将在从当前的显示位置开始，显示定义在 SRAM 中的字符串（str 为 SRAM 中定义的字符串的指针）。

☺ void lcd_putsf(char flash *str)：该函数将在从当前的显示位置开始，显示定义在 Flash 中的字符串（str 为 Flash 中定义的字符串的指针）。

☺ void lcd_write_byte(unsigned char addr,unsigned char data)：该函数向 LCD 模块的字符发生器或显存写一字节 data。

任务 2　字符式 LCD 显示字符串

使用 HD44780 字符编码，在 SMC1602A 液晶上显示字符串，第 1 行显示字符为“GOOD”；第 2 行显示字符为“Proteus 7.10”。

硬件设计

在 Proteus 中没有型号为 SMC1602A 的字符式 LCD，但 LM016L 与 SMC1602A 完全兼容，因此可用 LM016L 进行替代。在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 10-6 所列的元器件，并完成图 10-5 所示的硬件电路图设计。

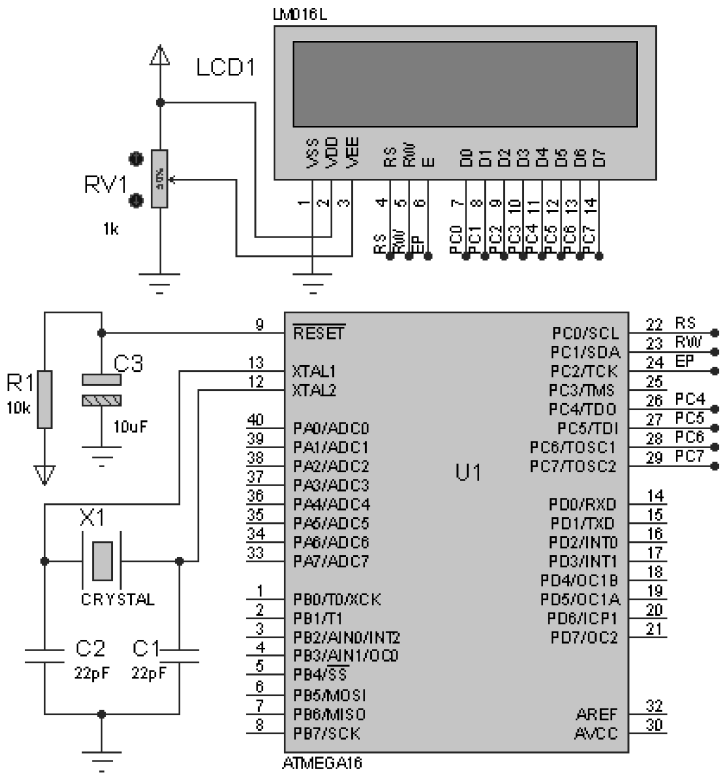


图 10-5　字符式 LCD 显示字符串电路图

表 10-6　字符式 LCD 显示字符串所使用的元器件

单片机 ATmega16	瓷片电容 CAP22pF	电阻 RES	电解电容 CAP-ELEC 10uF
晶振 CRYSTAL 8MHz	LM016L 字符式 LCD	可调电阻 POT-HG	

从表 10-1 中可以看出，字符串“GOOD”对应的字符编码分别为 0x47,0x4F, 0x4F,0x44, 0x00；字符串“Proteus7.10”对应的字符编码分别为 0x50,0x72,0x6F,0x74,0x65, 0x75,0x73, 0x37,0x2E,0x31,0x30。

软件设计

字符式 LCD 显示字符串时，首先要对 LCD 进行初始化，再分别确定第 1 行的显示起始坐标和第 2 行的显示起始坐标，最后分别将显示内容送到第 1 行和第 2 行即可。其程序流程图如图 10-6 所示。

源程序

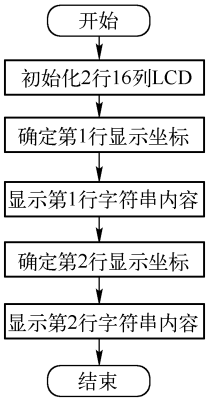


图 10-6　字符式 LCD 显示字符串程序流程图

```
/*****************************************************
  文件名:          字符式 LCD 显示字符串.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
  说明:            使用字符编码,第 1 行显示 GOOD   第 2 行显示
Proteus7.10
*****************************************************/
#include "mega16.h"
#include   "lcd.h"
#define uchar unsigned char
#define uint unsigned int
#asm
.equ __lcd_port=0x15;                                  //PORTC 数据寄存器地址
#endasm
typedef uchar byte;
flash   byte dis1[]={0x47,0x4F,0x4F,0x44,0x00};        //GOOD
flash   byte dis2[]={0x50,0x72,0x6F,0x74,0x65,
                     0x75,0x73,0x37,0x2E,0x31,0x30,0x00};   //Proteus7.10
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void main(void)
{
  char flash *str;
  lcd_init(16);                                //初始化 2 行 16 列 LCD
  while(1)
    {
      lcd_clear();                             //清屏 LCD，显示坐标设定在 0 列 0 行
      str = dis1;
      lcd_gotoxy(6,0);                         //设置当前坐标为 6 列第 1 行
      lcd_putsf(str);                          //在坐标中显示 Flash 中的字符串“GOOD”
      str = dis2;
      lcd_gotoxy(3,1);                         //设置下一个坐标为 3 列第 2 行
      lcd_putsf(str);                          //在坐标中显示 Flash 中的字符串“Proteus7.10”
      delay(200);
    }
}
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“字符式 LCD 显示字符串.COFF”文件，然后在 Proteus ISIS 中打开已创建的字符式 LCD 显示字符串电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试运行状态下，可看见在 LM016L 液晶上显示相应字符串，仿真效果如图 10-7 所示。对应表 10-1，更改程序中 dis1[]和 dis2[]的内容可显示不同的字符串。

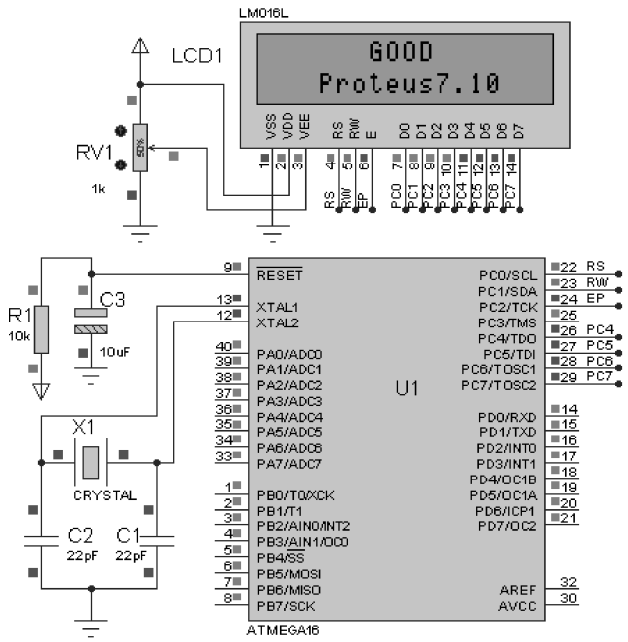


图 10-7　字符式 LCD 显示字符串仿真效果图

任务 3　字符式 LCD 显示汉字

使用 SMC1602A 液晶显示器显示两行汉字，第 1 行显示“天上人间”；第 2 行空一列后显示字符“copy：天上人间”。其中，第 2 行是每按一次按键显示一个字符串或汉字，即逐字显示。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 10-7 所列的元器件，并完成图 10-8 所示的硬件电路图设计。

表 10-7　字符式 LCD 显示汉字所使用的元器件

单片机 ATmega16	瓷片电容 CAP22pF	电阻 RES	电解电容 CAP-ELEC 10uF
晶振 CRYSTAL 8MHz	LM016L 字符式 LCD	可调电阻 POT-HG	按钮 BUTTON

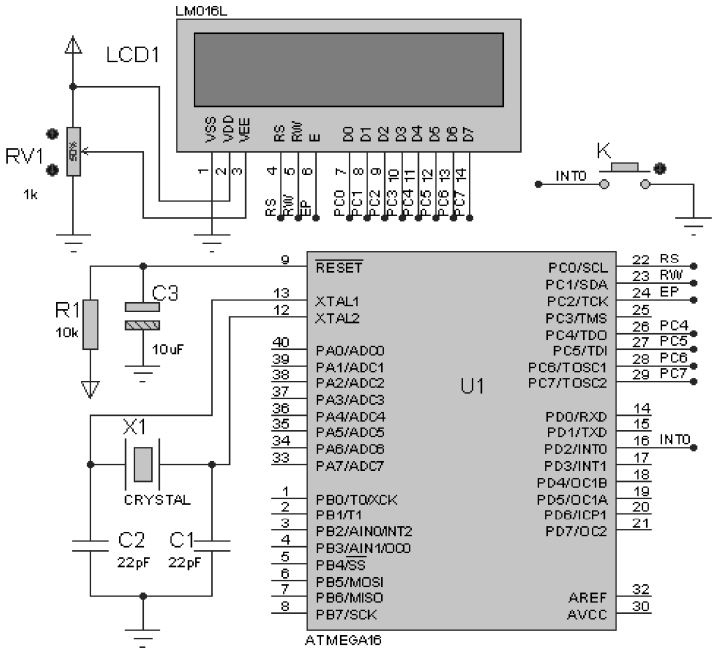


图 10-8　字符式 LCD 显示汉字原理图

软件设计

在 CodeVisionAVR 中，允许用户自己设计、定义和使用特殊的符号和图形。在 HD44780 中没有内置“天上人间”这 4 个汉字，要显示这些汉字，需要用户自己先对其定义，再使用 lcd_write_byte 和 lcd_putchar 指令即可。HD44780 支持用户自己最多定义 8 个 5×8 点阵的字符和图形，这些汉字点阵如图 10-9 所示（图中汉字采用了 x 轴镜像方式）。由于每个汉字为 5×8 点阵，采用 9 个段码时，每个段码为 8 位，而实际为 5 位，要形成 8 位，则每个段码的高 3 位补 0，因此这几个汉字的字模段码值见表 10-8。其程序流程图如图 10-10 所示。

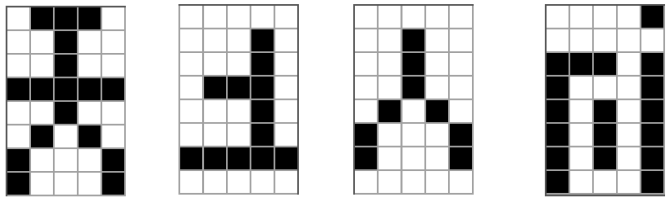


图 10-9 “天上人间”汉字点阵

表 10-8 “天上人间”字模段码值

显示顺序	字　符	段　码
0	"天"	0x0E,0x04,0x04,0x1F,0x04,0x0A,0x11,0x11
1	"上"	0x00,0x08,0x08,0x0E,0x08,0x08,0x1F,0x00
2	"人"	0x00,0x04,0x04,0x04,0x0A,0x11,0x11,0x00
3	"间"	0x10,0x00,0x17,0x11,0x15,0x15,0x15,0x11

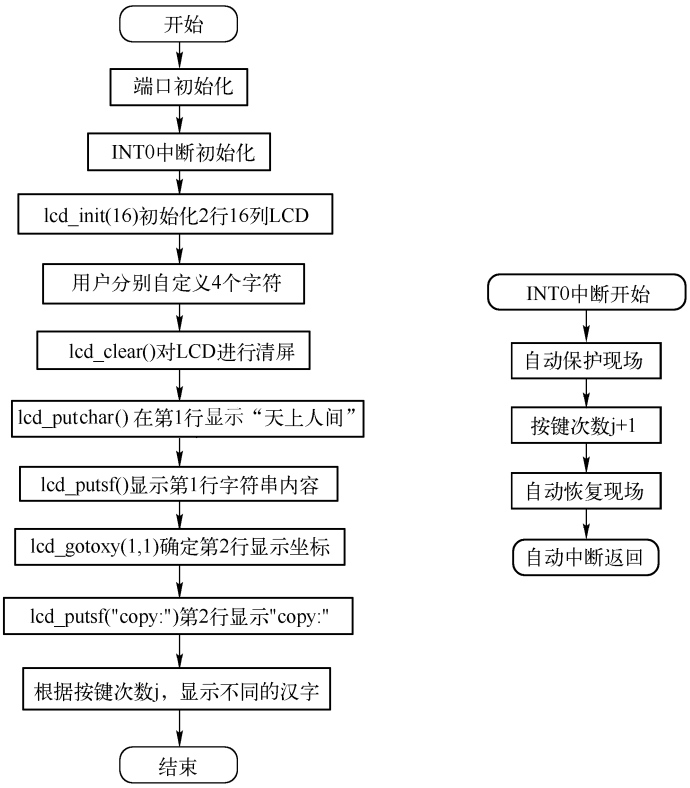


图 10-10　字符式 LCD 显示汉字程序流程图

源程序

```
/*****************************************************
  文件名:          字符式 LCD 显示汉字.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
  说明:            使用字符编码,第 1 行显示 天上人间   第 2 行显示 copy:天上人间
*****************************************************/
#include "mega16.h"
#include   "lcd.h"
#define   uchar unsigned char
#define   uint unsigned int
#define   button   PIND.2
#asm
.equ __lcd_port=0x15;                                    //PORTC 数据寄存器地址
#endasm
typedef uchar byte;
uint   cnt;
flash   byte dis[]={0x63,0x6F,0x70,0x79,0x3A,0x00};      //copy:
flash   byte char0[8]={0x0E,0x04,0x04,0x1F,0x04,0x0A,0x11,0x11};   /*"天",0*/
flash   byte char1[8]={0x00,0x08,0x08,0x0E,0x08,0x08,0x1F,0x00};   /*"上",1*/
flash   byte char2[8]={0x00,0x04,0x04,0x04,0x0A,0x11,0x11,0x00};   /*"人",2*/
flash   byte char3[8]={0x10,0x00,0x17,0x11,0x15,0x15,0x15,0x11};   /*"间",3*/
```

```
char flash *str;
void delay(uint k)
{
   uint m,n;
   for(m=0;m<k;m++)
      {
         for(n=0;n<1140;n++)
          {;}
      }
}
void define_char(byte flash *pc,byte char_code)                    //向 LCD 模块写一个用户定义的汉字
{
      byte i,a;
      a=(char_code<<3) | 0x40;
      for (i=0; i<8; i++)
        {
          lcd_write_byte(a++,*pc++);
        }
}
void display1(void)
{
   define_char(char0,0);                                           //自定义字符 0
   define_char(char1,1);                                           //自定义字符 1
   define_char(char2,2);                                           //自定义字符 2
   define_char(char3,3);                                           //自定义字符 3
   lcd_clear();                                                    //清屏 LCD，显示坐标设定在 0 列 0 行
   lcd_putchar(0);                                                 //显示自定义字符 0   天
   lcd_putchar(1);                                                 //显示自定义字符 1   上
   lcd_putchar(2);                                                 //显示自定义字符 2   人
   lcd_putchar(3);                                                 //显示自定义字符 3   间
   lcd_gotoxy(1,1);                                                //设置下一个坐标为 1 列第 2 行，即第 2 行
                                                                   //第 0 列空白显示
   lcd_putsf(str);                                                 //在当前坐标显示字符串"copy:"
}
void display2(void)
{
    switch(cnt)
      {
         case   1:                                                 //第 2 次按下按键时
            {
               lcd_gotoxy(1,1);
               lcd_putsf(str);
               break;
            }
         case   2:                                                 //第 2 次按下按键时
            {
               lcd_gotoxy(6,1);
               lcd_putchar(0);                                     //第 2 行显示"copy:天"
```

```
                break;
            }
          case   3:                                    //第 3 次按下按键时
            {
                lcd_gotoxy(6,1);
                lcd_putchar(0);
                lcd_putchar(1);                        //第 2 行显示"copy:天上"
                break;
            }
           case   4:                                   //第 4 次按下按键时
            {
                lcd_gotoxy(6,1);
                lcd_putchar(0);
                lcd_putchar(1);
                lcd_putchar(2);                        //第 2 行显示"copy:天上人"
                break;
            }
           case   5:                                   //第 5 次按下按键时
            {
                lcd_gotoxy(6,1);
                lcd_putchar(0);
                lcd_putchar(1);
                lcd_putchar(2);
                lcd_putchar(3);                        //第 2 行显示"copy:天上人间"
                break;
            }
           case   6:                                   //第 6 次按下按键时
            {
                lcd_clear();
                lcd_gotoxy(0,0);
                lcd_putchar(0);
                lcd_putchar(1);
                lcd_putchar(2);
                lcd_putchar(3);
                lcd_gotoxy(1,1);
                cnt=0;
                break;
            }
      }
}
interrupt[EXT_INT0] void int0_isr(void)                //INT0 中断函数
{
  delay(10);
  if(button= =0)                                       //再次确认按键，软件防抖
     {
       cnt++;
     }
}
void INT0_init(void)                                   //中断初始化设置
{
   MCUCR=0x02;                                         //INT0 为下降沿时产生中断请求
    GICR=0x40;                                         //允许 INT0 产生中断
```

```
    #asm("sei")                                    //使能全局中断
}
void port_init(void)                               //端口初始化
{
     DDRD=0xFB;
    PORTD=0xFF;
}
void main(void)
{
    lcd_init(16);                                  //初始化 2 行 16 列 LCD
    port_init();
    INT0_init();
    str = dis;
    display1();
    while(1)
      {
        display2();
      }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“字符式 LCD 显示汉字.COFF”文件，然后在 Proteus ISIS 中打开已创建的字符式 LCD 显示汉字电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试运行状态下，可看见字符式 LCD 的第 1 行显示“天上人间”；第 2 行的汉字为逐字显示，仿真效果如图 10-11 所示。

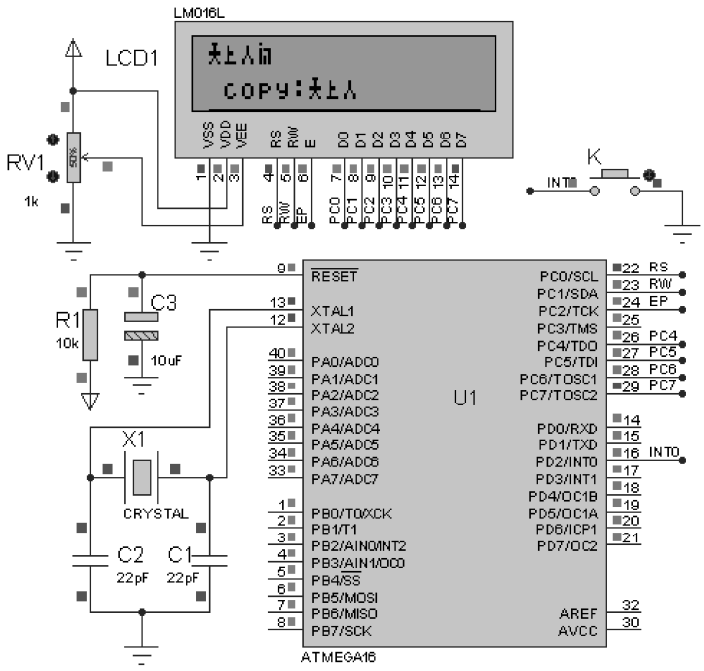


图 10-11　字符式 LCD 显示汉字仿真效果图

任务 4　字符式 LCD 显示可调数字钟

使用 SMC1602A 液晶显示器显示两行字符，第 1 行显示字符串“Digital clock” ”；第 2 行移位显示“time:”和时间，时间的初始值为“08-28-00。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 10-10 所列的元器件，并完成图 10-12 所示的硬件电路图设计。

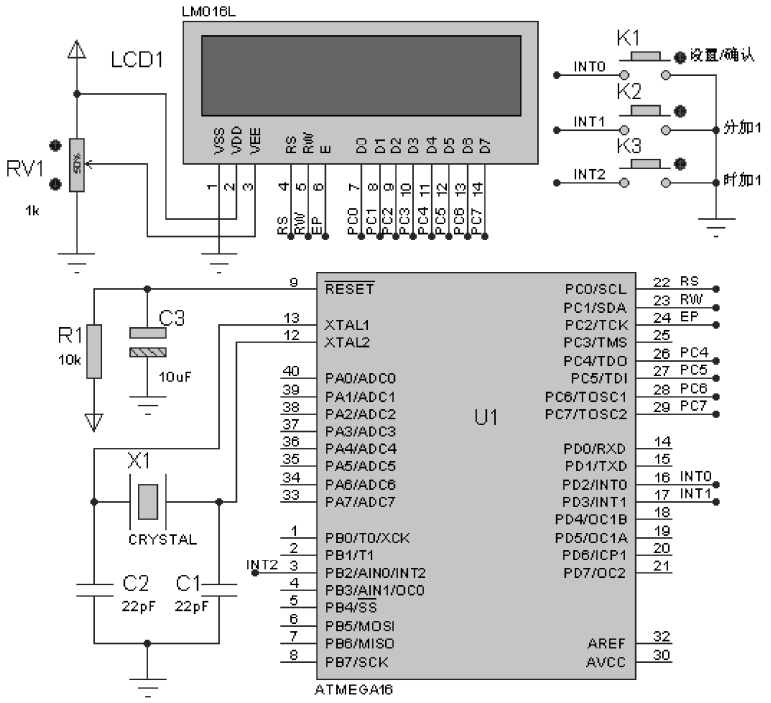


图 10-12　字符式 LCD 显示可调数字钟电路图

软件设计

要显示时间，必须要有计时。在该程序中，可利用 T/C0 来实现。在主程序中先进行端口、外部中断、T/C0、LCD 等初始化设置，并启动相应中断，然后将时间值进行相应整定，并将这些值送入 LCD 使其显示出来，其程序流程图如图 10-13 所示。

若设置 T/C0 的分频系数为 1024，系统时钟为 8MHz 时，需计数 30 次（1s/32.768ms≈30），因此，T/C0 初始化时相关寄存器设置为 TCCR0=0b00000101=0x05，TCNT0=0x00，OCR0=0x00，TIMSK=0x01。

对于外部中断而言，在初始化时，GICR=0x40，允许 INT0 中断，而 INT1 和 INT2 禁止中断。在 INT0 中断函数中，第 1 次按下按键 K1 时，GICR=0xE0，开启 INT1 和 INT2 中断，即允许调分和调时操作，而 TIMSK=0x00，暂停 T/C0 计时。第 2 次按下按键 K1 时，

GICR=0x40，允许 INT0 中断，而 INT1 和 INT2 禁止中断，即禁止调分和调时操作，而 TIMSK=0x01，启动 T/C0 计时。INT1 和 INT2 中断函数分别对分和时执行加 1 操作。

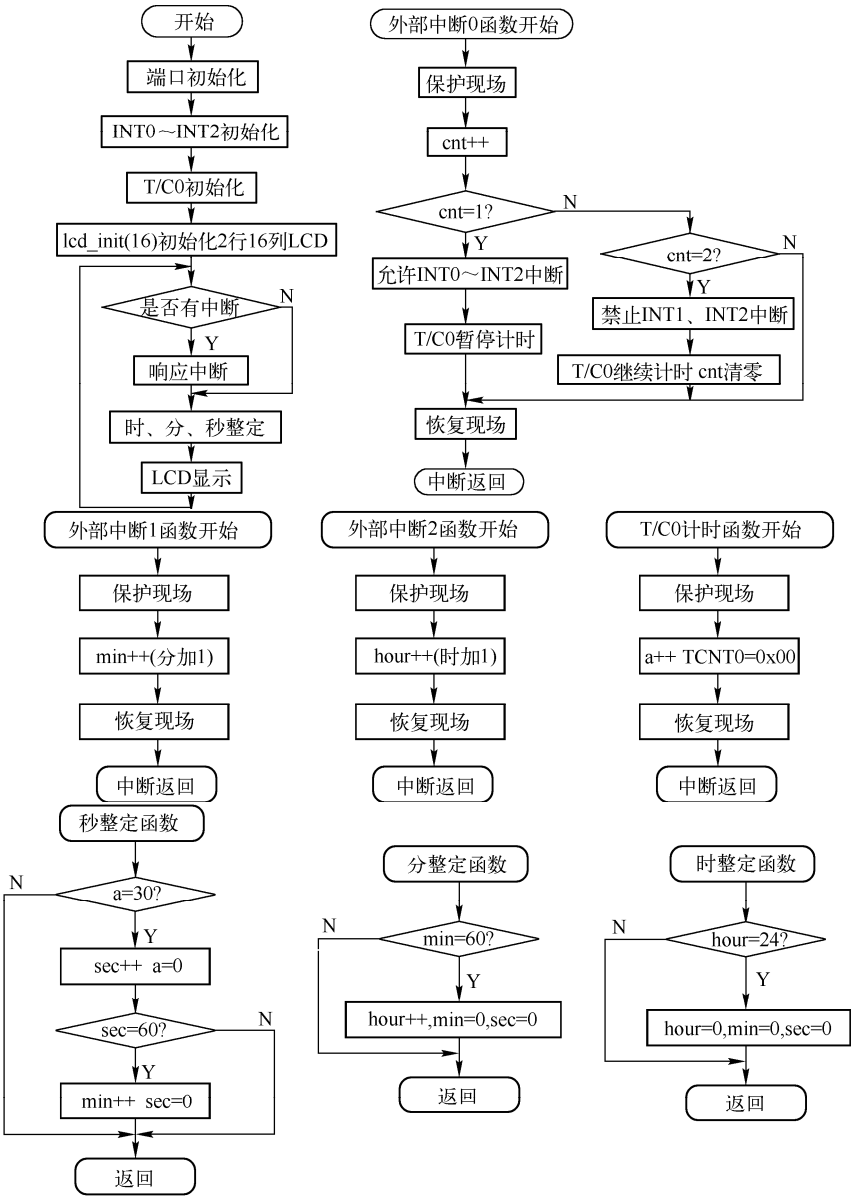


图 10-13　字符式 LCD 显示可调数字钟程序流程图

除完成初始化外，还需要时、分、秒的调整函数，以及 LCD 时间显示整定函数和 LCD 显示函数。在秒调整函数中，T/C0 每定时达 1s 时，sec 加 1。如果 sec 达到 60，则 sec 清零，而 min 加 1。在分调整函数中，如果 min 达到 60，则 min、sec 清零，而 hour 加 1。在时调整函数中，如果 hour 达到 24，则 hour、min、sec 均清零。LCD 时间显示整定函数用于将时、分、秒的高、低位进行分离，并送入相应的存储空间。在 LCD 显示函数中，首先使用 lcd_gotoxy(0,0)指令确定第 1 行显示坐标，并使用 lcd_putsf(str)指令显示第 1 行的内容，然后依此方法显示第 2 行的内容。

源程序

```
/*****************************************************
 文件名:          字符式 LCD 显示可调数字钟.c
 单片机型号:      ATmega16
 晶振频率:        8.0MHz
 说明:            直接使用字符，第 1 行显示 Digital clock  第二行显示“time:”和时间值
*****************************************************/
#include "mega16.h"
#include   "lcd.h"
#define   uchar unsigned char
#define   uint unsigned int
#define   button1   PIND.2                    //K1 设置/确认键
#define   button2   PIND.3                    //K2 分加 1
#define   button3   PINB.2                    //K3 时加 1
#asm
.equ __lcd_port=0x15;                         //PORTC 数据寄存器地址
#endasm
typedef uchar byte;
uint     cnt;                                 //K1 按键次数
uint   hour=8,min=18,sec;                     //时分秒定义
uint     a;
flash char dis_str[]=" Digital   clock ";      //直接使用字符
uchar lcdd[8];
char flash *str;
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
interrupt[EXT_INT0] void int0_isr(void)       //INT0 中断函数
{
  delay(10);
  if(button1= =0)                             //再次确认按键，软件防抖
    {
      cnt++;
      if(cnt= =1)
        {
          GICR=0xE0;                          //允许 INT0、INT1、INT2 产生中断
          TIMSK=0x00;                         //T/C0 停止计时
        }
      if(cnt>=2)
        {
          GICR=0x40;                          //允许 INT0 中断，INT1、INT2 中断禁止
          TIMSK=0x01;                         //T/C0 启动计时
          cnt=0;                              //K1 按下值复位
        }
```

```
    }
}
interrupt[EXT_INT1] void int1_isr(void)                    //INT1 中断函数
{
  delay(10);
  if(button2= =0)                                          //再次确认按键，软件防抖
    {
      min++;                                               //分加 1
    }
}
interrupt[EXT_INT2] void int2_isr(void)                    //INT2 中断函数
{
  delay(10);
  if(button3= =0)                                          //再次确认按键，软件防抖
    {
      hour++;                                              //时加 1
    }
}
interrupt [TIM0_OVF] void timer0_ovf_isr(void)             //T/C0 产生秒
{
  a++;
  TCNT0=0x00;
}
void   sec_in(void)                                        //秒调整
{
  if(a= =30)
    {
      sec++;
      a=0;
      if(sec= =60)
        {
          sec=0;
          min++;
        }
    }
}
void min_in(void)                                          //分调整
{
  if(min= =60)
    {
      hour++;
      min=0;
      sec=0;
    }
}
void hour_in(void)                                         //时调整
{
   if(hour= =24)
    {
      hour=0;
      min=0;
      sec=0;
```

```
    }
}
void lcd_in(void)                                    //LCD 时间显示整定
{
  lcdd[7]=sec%10+0x30;
  lcdd[6]=sec/10+0x30;
  lcdd[5]=0x2d;
  lcdd[4]=min%10+0x30;
  lcdd[3]=min/10+0x30;
  lcdd[2]=0x2d;
  lcdd[1]=hour%10+0x30;
  lcdd[0]=hour/10+0x30;
}
void display(void)
{
   uint i;
   str=dis_str;
   lcd_in();
   lcd_gotoxy(0,0);
   lcd_putsf(str);
   lcd_gotoxy(2,1);
   lcd_putsf("time:");
   lcd_gotoxy(7,1);
   for(i=0;i<8;i++)
      {
        lcd_putchar(lcdd[i]);
      }
}
void INT_init(void)                                  //中断初始化设置
{
   MCUCR=0x0A;                                       //定义 INT0 和 INT1 为下降沿时产生中断
   MCUCSR=0x00;                                      //定义 INT2 为下降沿时产生中断
    GICR=0x40;                                       //只允许 INT0 产生中断
    #asm("sei")                                      //开启总中断
}
void TC0_init(void)                                  //T/C0 初始化设置
{
  TCCR0=0x05;       //WGM01=0、WGM00=0，选择普通模式；COM01=0、COM00=0，使
                    //PB3 为通用 I/O（OC0 不与引脚相连）；CS02=1、CS01=0、CS01=1，选
                    //择 1024 分频
  TCNT0=0x00;
   OCR0=0x00;
  TIMSK=0x01;                                        //T/C0 启动计时
}
void port_init(void)
{
     DDRB=0xFB;
   PORTC=0xFF;
     DDRD=0xF3;
   PORTD=0xFF;
}
```

```
void main(void)
{
   port_init();
   INT_init();
   TC0_init();
   lcd_init(16);
   while(1)
      {
          sec_in();
          min_in();
          hour_in();
          display();
      }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“字符式 LCD 显示可调数字钟.COFF”文件，然后在 Proteus ISIS 中打开已创建的字符式 LCD 显示可调数字钟电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试运行状态下，LCD 显示的时间初值为 08-18-00，而后在此基础上进行计时。第 1 次按下按键 K1 后，可按下按键 K2 或 K3 进行分、时调整，第 2 次按下按键 K1 后，在调整好时间的基础上进行计时，其仿真效果如图 10-14 所示。

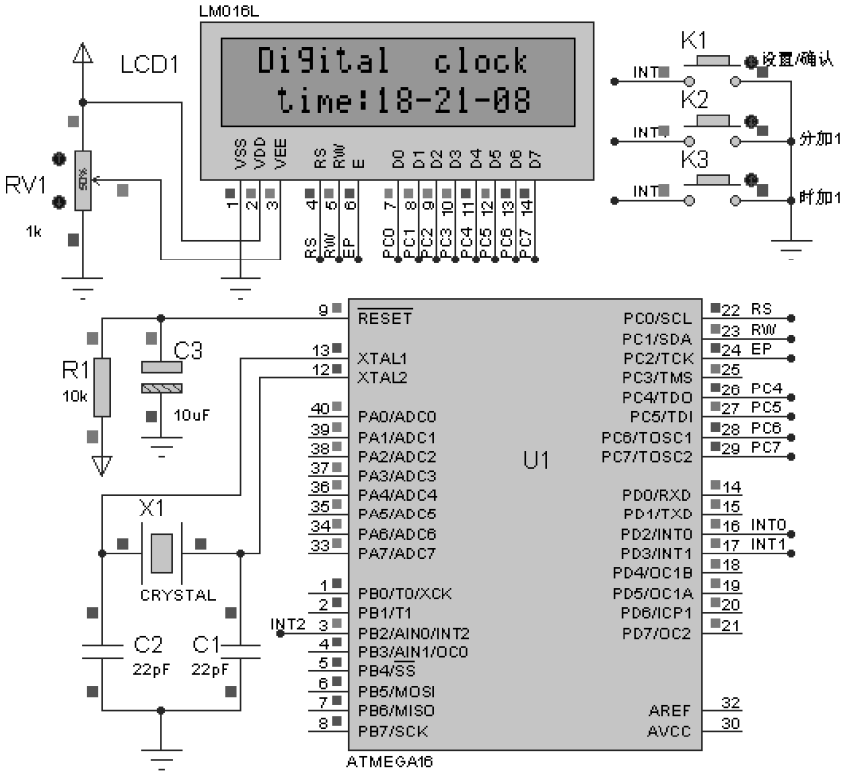


图 10-14　字符式 LCD 显示可调数字钟仿真效果图

任务 5　字符式 LCD 滚动显示

使用 HD44780 内置字符集，在 SMC1602A 液晶上滚动显示 4 组字符串，第 1 组字符串为“czpmcu@126.com”；第 2 组显示字符串为“QQ:769879416”； 第 3 组字符串为“2013-01-23”；第 4 组显示字符串为“ATmega16-DIL40”。



硬件设计

本任务采用任务 2 中图 10-5 所示的硬件电路图。



软件设计

本任务是在任务 3 的基础上进行的，要实现向上滚动显示的效果，可以认为是多屏循环显示的效果，即第 1 屏的第 1 行显示第 0 组字符串（第 0 组字符串是根据需要而添加的，该字符串显示内容为空），第 2 行显示第 1 组字符串；第 2 屏的第 1 行显示第 1 组字符串，第 2 行显示第 2 组字符串；第 3 屏的第 1 行显示第 2 组字符串，第 2 行显示第 3 组字符串；第 4 屏的第 1 行显示第 3 组字符串，第 2 行显示第 4 组字符串；第 5 屏的第 1 行显示第 4 组字符串，第 2 行显示第 1 组字符串，然后重复执行第 2 屏至第 5 屏的显示效果即可。为了获得较好的显示效果，每屏显示时暂停一段时间，然后清屏。其程序流程图如图 10-15 所示。

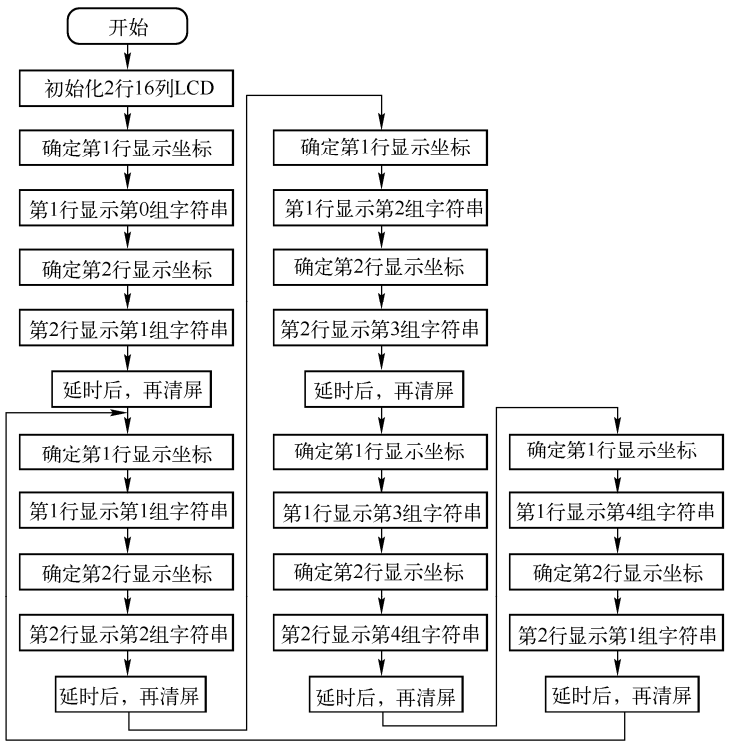


图 10-15　字符式 LCD 滚动显示程序流程图

源程序

```
/*****************************************************
  文件名:          字符式 LCD 滚动显示.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#include   "lcd.h"
#define   uchar unsigned char
#define   uint unsigned int
#asm
.equ __lcd_port=0x15;                              //PORTC 数据寄存器地址
#endasm
flash char dis_str0[]="                            ";
flash char dis_str1[]=" czpmcu@126.com ";
flash char dis_str2[]="   QQ:769879416    ";
flash char dis_str3[]="     2013-01-23     ";
flash char dis_str4[]=" ATmega16-DIL40 ";
char flash *str;
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void display(void)
{
    lcd_clear();                                   //LCD 清屏
    str=dis_str0;
    lcd_gotoxy(0,0);                               //在 0 行 0 列显示 dis_str0[]
    lcd_putsf(str);
    str=dis_str1;
    lcd_gotoxy(0,1);                               //在 1 行 0 列显示 dis_str1[]
    lcd_putsf(str);
    delay(1000);
    lcd_clear();                                   //LCD 清屏
    str=dis_str1;
    lcd_gotoxy(0,0);                               //在 0 行 0 列显示 dis_str1[]
    lcd_putsf(str);
    str=dis_str2;
```

```
    lcd_gotoxy(0,1);                    //在 1 行 0 列显示 dis_str2[]
    lcd_putsf(str);
    delay(1000);
    lcd_clear();                        //LCD 清屏
    str=dis_str2;
    lcd_gotoxy(0,0);                    //在 0 行 0 列显示 dis_str2[]
    lcd_putsf(str);
    str=dis_str3;
    lcd_gotoxy(0,1);                    //在 1 行 0 列显示 dis_str3[]
    lcd_putsf(str);
    delay(1000);
    lcd_clear();                        //LCD 清屏
    str=dis_str3;
    lcd_gotoxy(0,0);                    //在 0 行 0 列显示 dis_str3[]
    lcd_putsf(str);
    str=dis_str4;
    lcd_gotoxy(0,1);                    //在 1 行 0 列显示 dis_str4[]
    lcd_putsf(str);
    delay(1000);
    lcd_clear();                        //LCD 清屏
    str=dis_str4;
    lcd_gotoxy(0,0);                    //在 0 行 0 列显示 dis_str4[]
    lcd_putsf(str);
    str=dis_str1;
    lcd_gotoxy(0,1);                    //在 1 行 0 列显示 dis_str1[]
    lcd_putsf(str);
    delay(2000);
}
void main(void)
{
    lcd_init(16);
    while(1)
      {
        display();
      }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“字符式 LCD 滚动显示.COFF”文件，然后在 Proteus ISIS 中打开已创建的字符式 LCD 显示滚动显示电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试运行状态下可看见在 LM016L 液晶上字符串向上滚动显示，仿真效果如图 10-16 所示。

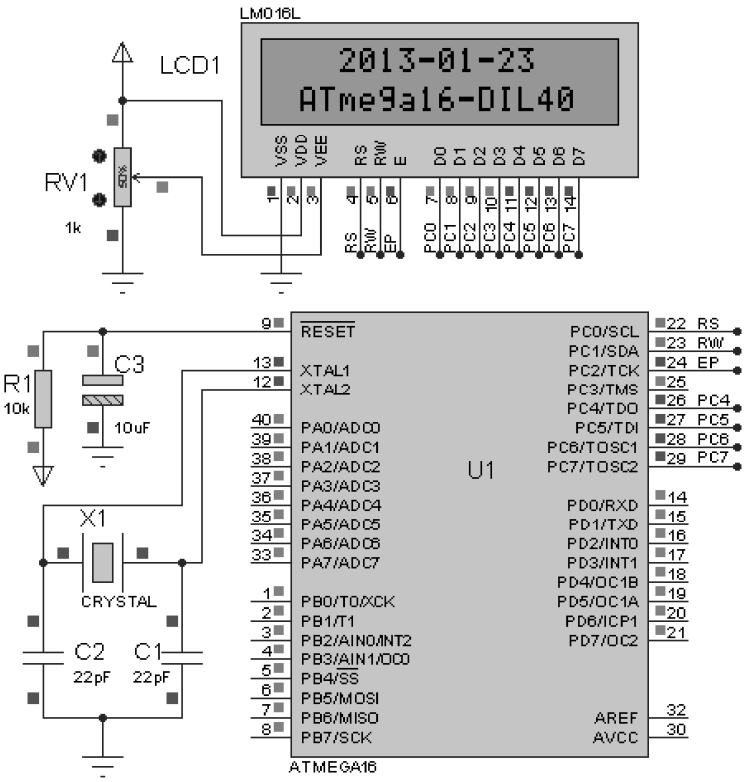


图 10-16　字符式 LCD 滚动显示仿真效果图

任务 6　字符式 LCD 移位显示

使用 HD44780 内置字符集，在 SMC1602A 液晶上进行相应显示。要求在默认情况下，闪屏显示；按下按键 K2 后，进入滚动显示；按下按键 K3 时，进入滚动显示；按下按键 K1 时，返回到默认情况下的第 1 屏显示。



硬件设计

本任务采用任务 4 中图 10-12 所示的硬件电路图。



软件设计

本任务由 3 个按键进行界面显示模式控制，因此可以使用 3 个外部中断 INT0～INT2 来控制 cnt 值。根据 cnt 值的不同，选择不同的显示界面，即 INT0 有效，cnt=0，选择闪屏显示；INT1 有效，cnt=1,选择滚动显示；INT2 有效，cnt=2，选择移位显示。

在闪屏显示控制中，主要是利用清屏指令来完成的；滚动显示控制可参考任务 5；移位显示最好使用另一个变量来指向 Flash，因此程序中需定义 str1 和 str2 两个变量，其中 str2 专用于移位控制。移位字符显示完后，应将移位字符串重新赋值给 str2，为下轮移位做好准备。其程序流程图如图 10-17 所示。

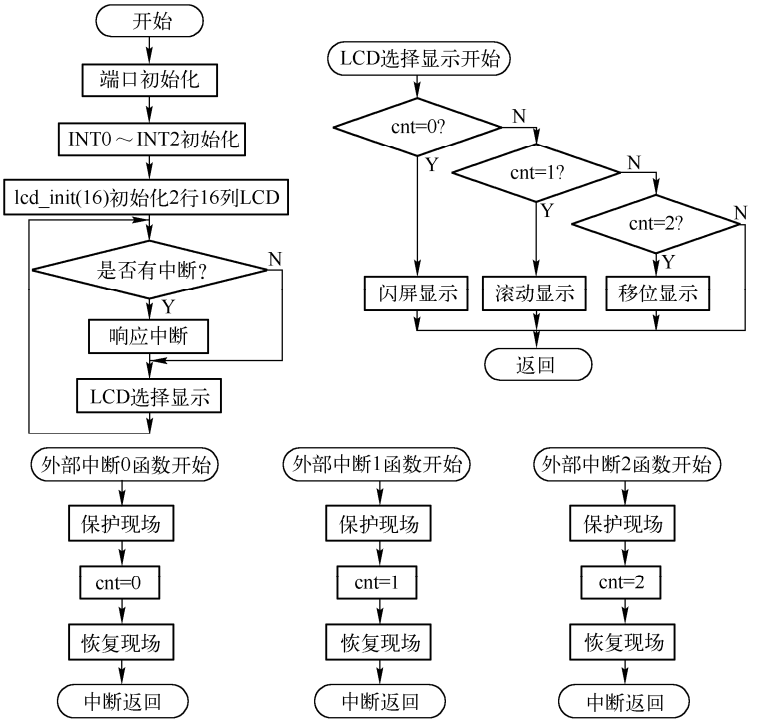


图 10-17　字符式 LCD 移位显示程序流程图

源程序

```
/*****************************************************
  文件名:          字符式 LCD 移位显示.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#include   "lcd.h"
#define   uchar unsigned char
#define   uint unsigned int
#define   button1   PIND.2                          //K1 设置/确认键
#define   button2   PIND.3                          //K2 分加 1
#define   button3   PINB.2                          //K3 时加 1
#asm
.equ __lcd_port=0x15;                               //PORTC 数据寄存器地址
#endasm
typedef uchar byte;
uint     cnt;
flash char dis_str0[]=" Main interface ";
flash char dis_str1[]=" Select   button ";
flash char dis_str2[]="                 ";
flash char dis_str3[]=" Welcome   Key2   ";
flash char dis_str4[]="Scrollingdisplay";
flash char dis_str5[]="   Key1   Return   ";
```

```
flash char dis_str6[]=" Welcome    Key3    ";
flash char dis_str7[]="Shift display Key1    Return ";          //移位字符串
char flash *str1;
char flash *str2;                                               //定义 str2 用于移位显示
void delay(uint k)
{
   uint m,n;
   for(m=0;m<k;m++)
      {
         for(n=0;n<1140;n++)
           {;}
      }
}
interrupt[EXT_INT0] void int0_isr(void)                         //INT0 中断函数
{
   delay(10);
   if(button1= =0)                                              //再次确认按键，软件防抖
      {
         cnt=0;
      }
}
interrupt[EXT_INT1] void int1_isr(void)                         //INT1 中断函数
{
   delay(10);
   if(button2= =0)                                              //再次确认按键，软件防抖
      {
         cnt=1;
      }
}
interrupt[EXT_INT2] void int2_isr(void)                         //INT2 中断函数
{
   delay(10);
   if(button3= =0)                                              //再次确认按键，软件防抖
      {
         cnt=2;
      }
}
void display1(void)                                             //闪屏显示
{
    uint i;
    lcd_clear();                                                //LCD 清屏
    str1=dis_str0;
    lcd_gotoxy(0,0);                                            //在 0 行 0 列显示 dis_str0[]
    lcd_putsf(str1);
    str1=dis_str1;
    lcd_gotoxy(0,1);                                            //在 1 行 0 列显示 dis_str1[]
    lcd_putsf(str1);
    delay(2000);
```

```
    for (i=0;i<2;i++)
       {
         lcd_clear();                          //LCD 清屏
         delay(300);
         str1=dis_str0;
         lcd_gotoxy(0,0);                      //在 0 行 0 列显示 dis_str0[]
         lcd_putsf(str1);
         str1=dis_str1;
         lcd_gotoxy(0,1);                      //在 1 行 0 列显示 dis_str1[]
         lcd_putsf(str1);
         delay(300);
       }
}
void display2(void)                            //滚动显示
{
    lcd_clear();                               //LCD 清屏
    str1=dis_str2;
    lcd_gotoxy(0,0);                           //在 0 行 0 列显示 dis_str0[]
    lcd_putsf(str1);
    str1=dis_str3;
    lcd_gotoxy(0,1);                           //在 1 行 0 列显示 dis_str1[]
    lcd_putsf(str1);
    delay(500);
    lcd_clear();                               //LCD 清屏
    str1=dis_str3;
    lcd_gotoxy(0,0);                           //在 0 行 0 列显示 dis_str1[]
    lcd_putsf(str1);
    str1=dis_str4;
    lcd_gotoxy(0,1);                           //在 1 行 0 列显示 dis_str2[]
    lcd_putsf(str1);
    delay(500);
    lcd_clear();                               //LCD 清屏
    str1=dis_str4;
    lcd_gotoxy(0,0);                           //在 0 行 0 列显示 dis_str2[]
    lcd_putsf(str1);
    str1=dis_str5;
    lcd_gotoxy(0,1);                           //在 1 行 0 列显示 dis_str3[]
    lcd_putsf(str1);
    delay(500);
    lcd_clear();                               //LCD 清屏
    str1=dis_str5;
    lcd_gotoxy(0,0);                           //在 0 行 0 列显示 dis_str3[]
    lcd_putsf(str1);
    str1=dis_str3;
    lcd_gotoxy(0,1);                           //在 1 行 0 列显示 dis_str4[]
    lcd_putsf(str1);
    delay(500);
}
```

```
void display3(void)                          //移位显示
{
    lcd_clear();                             //LCD 清屏
    str1=dis_str6;
    lcd_gotoxy(0,0);                         //在 0 行 0 列显示 dis_str0[]
    lcd_putsf(str1);
    lcd_gotoxy(0,1);                         //在 1 行 0 列显示 dis_str1[]
    lcd_putsf(str2);
    if(*str2++= =0)
      {
        str2=dis_str7;                       //为下轮移位准备
      }
    delay(200);
}
void display(void)
{
    switch (cnt)
      {
        case 0:display1();break;
        case 1:display2();break;
        case 2:display3();break;
      }
}
void INT_init(void)                          //中断初始化设置
{
    MCUCR=0x0A;                              //定义 INT0 和 INT1 为下降沿时产
                                             //生中断
    MCUCSR=0x00;                             //定义 INT2 为下降沿时产生中断
     GICR=0xE0;                              //允许 INT0 产生中断
     #asm("sei")                             //开启总中断
}
void port_init(void)
{
     DDRB=0xFB;
    PORTB=0xFF;
     DDRD=0xF3;
    PORTD=0xFF;
}
void main(void)
{
  port_init();
  INT_init();
  lcd_init(16);
  str2=dis_str7;
  while(1)
    {
      display();
    }
}
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“字符式 LCD 移位显示.COFF”文件，然后在 Proteus ISIS 中打开已创建的字符式 LCD 显示移位显示电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试运行状态下，没有按下按键 K2 或 K3 时，LM016L 进行闪屏显示；按下按键 K2 后，LM016L 液晶上字符串向上滚动显示；按下按键 K3 后，LM016L 液晶上第 2 行的字符串左移显示，其仿真效果如图 10-18 所示。

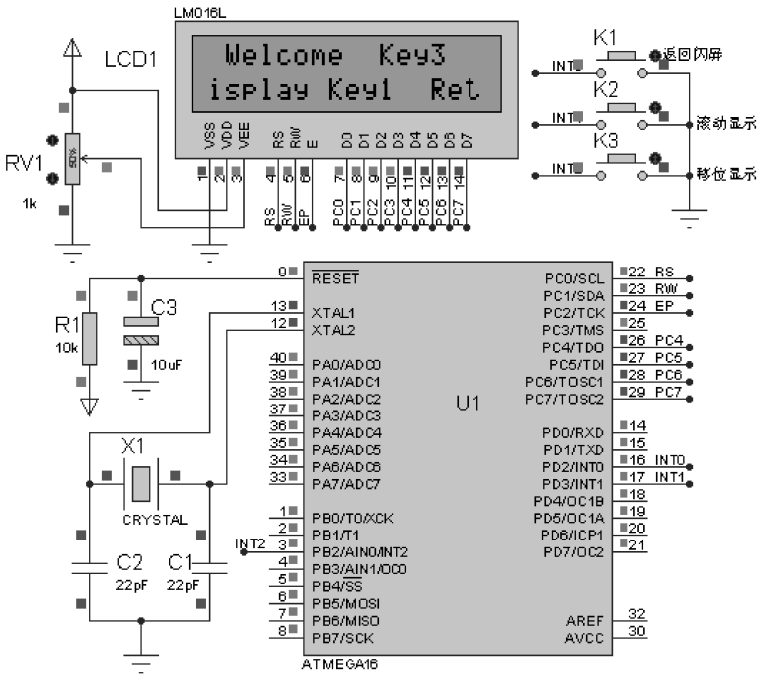


图 10-18　字符式 LCD 移位显示仿真效果图

项目十一　内置模拟比较器和 A/D 转换器的应用设计

〖知识目标〗

☺ 了解内置模拟比较器和 A/D 转换器的结构、控制寄存器定义及使用方法;

☺ 掌握模拟比较器、A/D 转换器的控制及应用。

〖能力目标〗

☺ 学会模拟比较器、A/D 转换器相关寄存器的设置;

☺ 会利用模拟比较器、A/D 转换器进行小系统的设计。

ATmega16 单片机中的模拟比较器和 A/D 转换器是支持模拟信号输入的功能接口。大部分 AVR 单片机通常具备这两种类型接口，而这样的功能接口在标准 8051 单片机中是没有的。

任务 1　内置模拟比较器和 A/D 转换器

1．ATmega16 模拟比较器

1）ATmega16 模拟比较器的结构　ATmega16 的模拟比较器可以实现对正极 AIN0 和负极 AIN1（分别对应于 ATmage16 的引脚 PB2、PB3）这两个输入端的模拟输入电压进行比较。当 AIN0 上的电压高于 AIN1 的电压时，模拟比较器输出 ACO 置“1”。比较器的输出还可以用于触发定时器/计数器 1 的输入捕获功能。此外，比较器的输出可以触发自己专有的、独立的模拟比较器中断。用户可以选择比较器是以上升沿、下降沿或交替变化的边沿作为模拟比较器中断的触发信号。ATmega16 单片机模拟比较器的结构如图 11-1 所示。

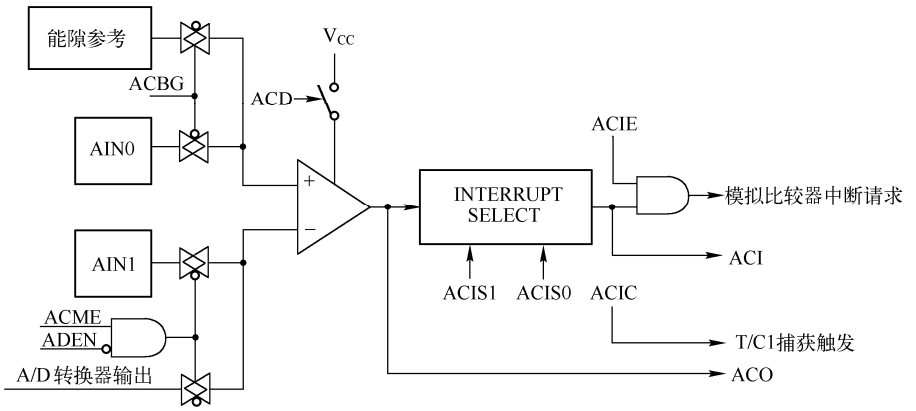


图 11-1　模拟比较器的结构框图

2）模拟比较器的相关寄存器

【特殊功能 I/O 寄存器（SFIOR）】

Bit	7	6	5	4	3	2	1	0
	ADTS2	ADTS1	ADTS0	-	ACME	PUD	PSR2	PSR10
读/写	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ ACME（bit3）：模拟比较器多路复用器使能控制位。若 ACME=1，并且 A/D 转换（ADC）功能被关闭（ADCSRA 寄存器中的 ADEN 使能位为“0”）时，允许使用 ADC 多路复用器选择 ADC 的模拟输入端口作为模拟比较器反相端的输入信号源；若 ACME=0 时，AIN1 引脚的信号将加到模拟比较器反相端上。

【模拟比较器控制和状态寄存器（ACSR）】

Bit	7	6	5	4	3	2	1	0
	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0
读/写	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ ACD（bit7）：模拟比较器禁用控制位。当 ACD=1 时，模拟比较器的电源被切断。该位可以在任何时候被置“1”，从而关闭模拟比较器，这样可以减少器件工作模式及空闲模式下的功耗。改变 ADC 位时，必须先将 ACSR 寄存器的 ACIE 位清零，禁止模拟比较器中断，否则 ACD 改变时可能会产生中断。

☺ ACBG（bit6）：模拟比较器的能隙参考源选择位。当 ACBG=1 时，芯片内部的一个固定能隙（Bandgap）参考电源 1.22V 将代替 AIN0 的输入，作为模拟比较器的正极输入端；当 ACBG=0 时，AIN0 的输入仍然作为模拟比较器的正极输入端。

☺ ACO（bit5）：模拟比较器输出。模拟比较器的输出信号经过同步处理后直接与 ACO 相连。由于经过了同步处理，所以 ACO 与模拟比较器的输出之间会有 1～2 个时钟的延时。

☺ ACI（bit4）：模拟比较器中断标志位。当模拟比较器的输出事件触发了由 ACIS1 和 ACIS0 定义的中断触发条件时，ACI 由硬件置“1”。若 ACIE 位置“1”，且状态寄存器中的 I 位为“1”时，MCU 响应模拟比较器中断。当 MCU 响应该中断时，ACI 会被硬件自动清空。此外，用户也可以对 ACI 标志位写入逻辑“1”来清零该位。

☺ ACIE（bit3）：模拟比较器中断允许控制位。若 ACIE=1 且状态寄存器中的 I 位为“1”时，允许模拟比较器中断触发；当 ACIE=0 时，禁止模拟比较器中断。

☺ ACIC（bit2）：模拟比较器输入捕获允许控制位。当 ACIC=1 时，定时器/计数器 1 的输入捕获功能将由模拟比较器的输出来触发。在这种情况下，模拟比较器的输出直接连到输入捕获前端逻辑电路上，从而能利用定时器/计数器 1 输入捕获中断的噪声消除和边缘选择的特性；当 ACIC=0 时，模拟比较器和输入捕获功能之间没有联系。为了使比较器可以触发定时器/计数器 1 的输入捕获中断，定时器中断屏蔽寄存器（TIMSK）中的 TICIE1 位必须置位。

☺ ACIS1、ACIS0（bit1、bit0）：模拟比较器中断模式选择位。这 2 位决定哪种模拟比较器的输出事件可以触发模拟比较器的中断。表 11-1 列出了不同的选择结果。

表 11-1　模拟比较器中断模式选择

ACIS1	ACIS0	中 断 模 式	ACIS1	ACIS0	中 断 模 式
0	0	比较器输出变化即可触发中断	1	0	比较器输出的下降沿触发中断
0	1	保留	1	1	比较器输出的上升沿触发中断

若要改变 ACIS1、ACIS0，必须先清除 ACSR 寄存器中的中断允许位，以禁止模拟比较器中断；否则，当这些位被改变时，会发生中断。

3）模拟比较器的多路输入　用户可以选择 ADC7～AC0 引脚中的任意一路的模拟信号代替 AIN1 引脚，作为模拟比较器的反相输入端。A/D 转换的 ADC 多路复用器提供这种选择的能力，当然，为了使用这个功能必须先关闭芯片的 ADC 功能。当模拟比较器的多路选择使能位（SFIOR 中的 ACME 位）置“1”，同时 ADC 被关闭时（ADCSRA 中的 ADEN 位置“0”），由寄存器 ADMUX 中的 MUX[2..0]位所确定的引脚将代替 AIN1 作为模拟比较器的反相输入端，见表 11-2。如果 ACME 被清零，或者 ADEN 被置 1，则 AIN1 仍将为模拟比较器的反相输入端。

表 11-2　模拟比较器多路输入选择

ACME	ADEN	MUX[2..0]	模拟比较器负极输入	ACME	ADEN	MUX[2..0]	模拟比较器负极输入
0	×	×××	AIN1	1	0	011	ADC3
1	1	×××	AIN1	1	0	100	ADC4
1	0	000	ADC0	1	0	101	ADC5
1	0	001	ADC1	1	0	110	ADC6
1	0	010	ADC2	1	0	111	ADC7

2．ATmega16 A/D 转器

能够将模拟量转换成数字量的器件称为 A/D 转换器 ADC（Analog to Digital Converter）。Atmega16 内部集成有 8 个 10 位精度的 ADC。

1）ATmega16 模数转器结构　ATmega16 的 ADC 与一个 8 位通道的模拟多路选择器连接，能够对来自 PORTA 端口的 8 路单端模拟输入电压进行采样，单端电压输入以 0V（GND）为参考。ADC 还支持 16 种差分电压输入组合，若差分输入电压的增益为 1×或 10×，则可获得 8 位的 ADC 转换精度。如果差分输入电压的增益为 200×，则 ADC 的转换精度为 7 位。

AVR 的 ADC 功能单元由独立的专用模拟电源引脚 AVcc 供电。AVcc 和 Vcc 的电压差别不能大于±0.3V。ADC 转换的参考电源可采用芯片内部的 2.56V 参考电源，或者采用 AVcc，也可使用外部参考电源。使用外部参考电源时，外部参考电源由引脚 AREF 接入。使用内部电压参考源时，可以通过在 AREF 引脚外部并接一个电容进行解耦，以提高 ADC 的抗噪性能。

ADC 功能单元包括采样保持电路，以确保输入电压在 A/D 转换过程中保持恒定。ADC 的功能单元框图如图 11-2 所示。

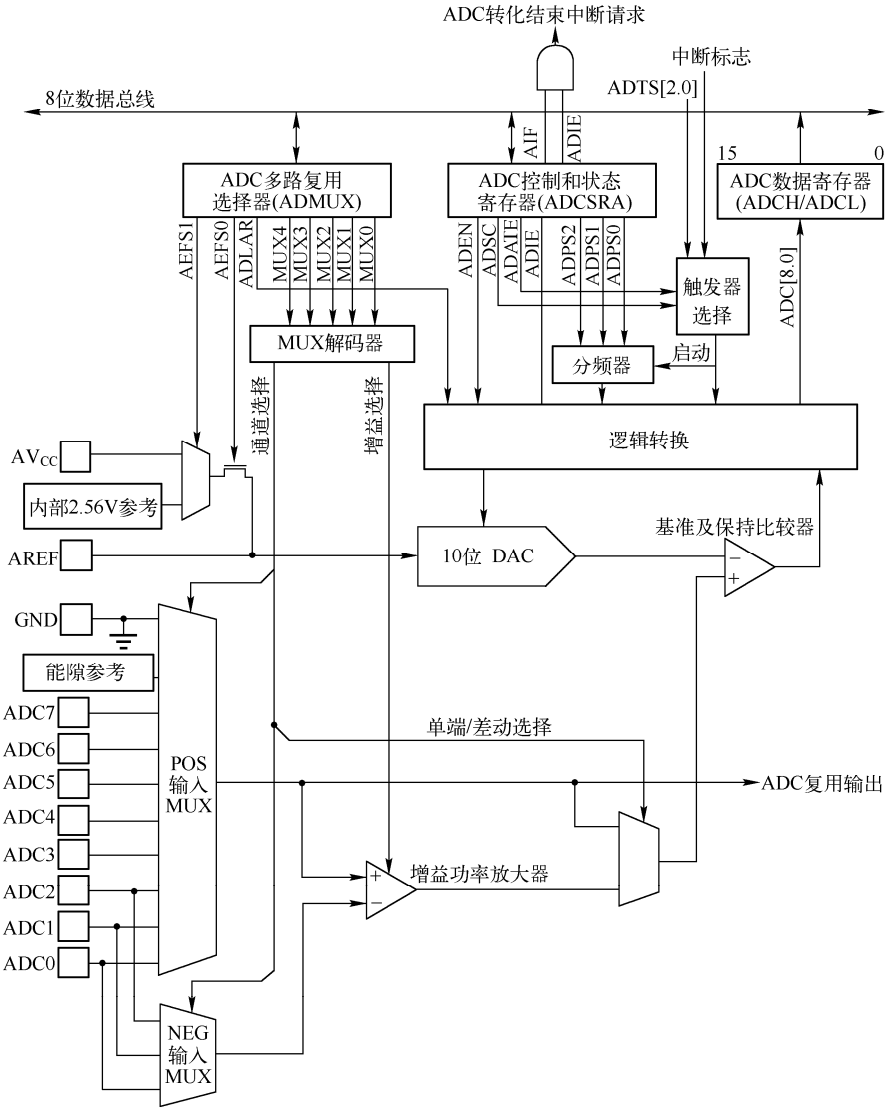


图 11-2　ADC 的功能单元框图

ADC 通过逐次比较（Successive Approximation）方式，将输入端的模拟电压转换成 10 位的数字量。最小值代表地，最大值为 AREF 引脚上的电压值减去 1LSB。通过对 ADMUX 寄存器中 REFSn 位的设置，可以将芯片内部参考电源（2.56V）或 AVcc 连接到 AREF 上，作为 A/D 转换的参考电压。此时，内部电压参考源可以通过在 AREF 引脚上外加电容进行解耦，以提高噪声抑制性能。

模拟输入通道和差分增益的选择是通过 ADMUX 寄存器中的 MUX 位设定的。任何一个 ADC 的输入引脚，包括地（GND）及内部的恒定能隙（fixed bandgap）电压参考源，都可以被选择用来作为 ADC 的单端输入信号。而 ADC 的某些输入引脚则可作为差分增益放大器的正、负极输入端。当选定了差分输入通道后，差分增益放大器先将两个输入通道上的电压差按选定增益系数放大，然后输入到 ADC 中。若选定使用单端输入通道，则增益放大器无效。

通过设置 ADCSRA 寄存器中的 ADC 使能位 ADEN 即可启动 ADC。在 ADEN 没有置“1”前，参考电压源和输入通道的选定将不起作用。当 ADEN 位清“0”后，ADC 将不消耗能量，因此建议在进入节电休眠模式前将 ADC 关掉。

ADC 将 10 位的转换结果放在 ADC 数据寄存器中（ADCH 和 ADCL）。默认情况下，转换结果为右端对齐（RIGHT ADJUSTED）。但可以通过设置 ADMUX 寄存器中的 ADLAR 位将其调整为左端对齐（LEFT ADJUSTED）。如果转换结果是左端对齐，并且只需要 8 位的精度，那么只需读取 ADCH 寄存器的数据作为转换结果就可达到要求了。否则，必须先读取 ADCL 寄存器，然后再读取 ADCH 寄存器，以保证数据寄存器中的内容是同一次转换的结果。因为一旦 ADCL 寄存器被读取，就阻断了 ADC 对 ADC 数据寄存器的操作。这就意味着一旦指令读取了 ADCL，必须紧接着读取一次 ADCH；如果在读取 ADCL 和读取 ADCH 的过程中正好有一次 A/D 转换完成，ADC 的 2 个数据寄存器的内容是不会被更新的，该次转换的结果就将丢失。只有当 ADCH 寄存器被读取后，ADC 才可以继续对 ADCL 和 ADCH 寄存器操作进行更新。

A/D 转换完成时可以触发中断，即使转换发生在读取 ADCH 与 ADCL 之间造成 ADC 无法访问数据寄存器，并因此丢失了转换数据，但仍会触发 ADC 中断。

2）A/D 转换器的相关寄存器

【ADC 多路复用器选择寄存器（ADMUX）】

Bit	7	6	5	4	3	2	1	0
	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ REFS1、REFS0（bit7、bit6）：ADC 参考电源选择位。这两位可以选择 ADC 的参考电源源，见表 11-3。如果在转换过程中改变了它们的设置，只有等到当前转换结束（ADCSRA 中的 ADIF 置位）后，该改变才会起作用。如果在 AREF 引脚上施加了外部参考电压，内部参考电压就不能被选用了。如果选择内部参考电源（AVcc、2.56V）作为 ADC 的参考电压，则 AREF 引脚上不得施加外部参考电压，只能在它与 GND 之间并接抗干扰电容。

表 11-3　ADC 参考电源的选择

REFS1	REFS0	ADC 参考电源	REFS1	REFS0	ADC 参考电源
0	0	断开内部参考电源连接，参考电源由外部引脚 AREF 输入	1	0	保留
0	1	AVcc，AREF 外部并接电容	1	1	内部 2.56V，AREF 外部并接电容

☺ ADLAR（bit5）：A/D 转换结果左对齐选择位。ADLAR 会影响 A/D 转换结果在 ADC 数据寄存器中的存放形式。当 ADLAR =1 时，将使转换结果左对齐，否则转换结果为右对齐。不论 ADC 是否正在进行转换，ADLAR 的改变将立即影响 ADC 数据寄存器的内容。

☺ MUX4 ~ MUX0（bit4 ~ bit0）：模拟通道的增益选择位。这 5 个位用于对连接到 ADC 的输入通道和差分通道的增益进行选择设置，见表 11-4。

注意:
只有当转换结束后（ADCSRA 的 ADIF 为“1”），改变这些位才有效。

表 11-4 ADC 输入通道的增益选择

MUX[4..0]	单端输入	差分正极输入	差分负极输入	增益
00000	ADC0	N/A		
00001	ADC1			
00010	ADC2			
00011	ADC3			
00100	ADC4			
00101	ADC5			
00110	ADC6			
00111	ADC7			
01000	N/A	ADC0	ADC0	10×
01001		ADC1	ADC0	10×
01010		ADC0	ADC0	200×
01011		ADC1	ADC0	200×
01100		ADC2	ADC2	10×
01101		ADC3	ADC2	10×
01110		ADC2	ADC2	200×
01111		ADC3	ADC2	200×
10000		ADC0	ADC1	1×
10001		ADC1	ADC1	1×
10010		ADC2	ADC1	1×
10011		ADC3	ADC1	1×
10100		ADC4	ADC1	1×
10101		ADC5	ADC1	1×
10110		ADC6	ADC1	1×
10111		ADC7	ADC1	1×
11000		ADC0	ADC2	1×
11001		ADC1	ADC2	1×
11010		ADC2	ADC2	1×
11011		ADC3	ADC2	1×
11100		ADC4	ADC2	1×
11101		ADC5	ADC2	1×
11110	1.22V(VBG)	N/A		
11111	0V(GND)			

【ADC 控制和状态寄存器 A（ADCSRA）】

bit	7	6	5	4	3	2	1	0
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ ADEN（bit7）：ADC 使能控制位。当 ADEN=1 时，使能 ADC；当 ADEN=0 时，关闭 ADC。如果在 A/D 转换过程中关闭 ADC，则立即中止正在进行的转换。

☺ ADSC（bit6）：ADC 开始转换位。在单次模式下，ADSC 置位将启动一次 A/D 转换。在连续转换模式下，ADSC 置位将启动首次转换。第 1 次转换（在 ADC 启动后，置位 ADSC，或者在使能 ADC 的同时置位 ADSC）需要 25 个 ADC 时钟周期，而不是正常情况下的 13 个。第 1 次转换会执行 ADC 初始化的工作。在 ADSC 转换的过程中，ADSC 将始终读出“1”。当转换完成后，它将改变为“0”，强制写入“0”是无效的。

☺ ADATE（bit5）：ADC 自动转换触发允许位。当 ADATE=1 时，允许 ADC 工作在自动转换触发工作模式下；当 ADATE=0 时，停止自动转换。在该模式下，ADC 不断对数据寄存器进行采样与更新。

☺ ADIF（bit4）：ADC 中断标志位。当 ADC 转换完成并且 ADC 数据寄存器更新后，该位置“1”。如果 ADIE 位和 SREG 中的全局中断使能位 I 也置“1”，则 A/D 转换结束中断服务程序立即执行，同时 ADIF 硬件清零。此外，还可以通过向此标志写“1”使 ADIF 位清零。

☺ ADIE（bit3）：ADC 中断允许位。如果 ADIE 位和 SREG 中的全局中断使能位 I 置“1”，则允许响应 A/D 转换完成中断。

☺ ADPS2 ~ ADPS0（bit2 ~ bit0）：ADC 时钟预分频选择位。这 3 位确定了 XTAL 与 ADC 输入时钟的分频系数，见表 11-5。

表 11-5　ADC 分频系数的选择

ADPS2	ADPS1	ADPS0	分频系数	ADPS2	ADPS1	ADPS0	分频系数
0	0	0	1	1	0	0	16
0	0	1	2	1	0	1	32
0	1	0	4	1	1	0	64
0	1	1	8	1	1	1	128

【ADC 数据寄存器（ADCH 和 ADCL）】ADC 数据寄存器用于存储 ADC 的转换结果，它由 ADCH 和 ADCL 两个 8 位寄存器构成。当 A/D 转换完成后，可以读取 ADC 寄存器的 ADC0 ~ ADC9，以得到 ADC 的转换结果。如果是差分输入，转换值为二进制数的补码形式。

一旦开始读取 ADCL 后，ADC 数据寄存器就不能被 ADC 更新了，直到 ADCH 寄存器被读取为止。因此，如果结果是左对齐（ADLAR=1）的，且不需要大于 8 位的精度，则仅读取 ADCH 寄存器就足够了。否则，必须先读取 ADCL 寄存器，再读取 ADCH 寄存器。

ADMUX 寄存器中的 ADLAR 位决定了从 ADC 数据寄存器中读取结果的格式。如果 ADLAR 位为“1”，结果将是左对齐的；如果 ADLAR 位为“0”（默认情况），结果将是右对齐的。

☺ 当 ADLR=0，A/D 转换结果右对齐时，ADC 结果的保存方式为

bit	15	14	13	12	11	10	9	8	
	—	—	—	—	—	—	ADC9	ADC8	ADCH
读/写	R	R	R	R	R	R	R	R	
初始值	0	0	0	0	0	0	0	0	
bit	7	6	5	4	3	2	1	0	
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
读/写	R	R	R	R	R	R	R	R	
初始值	0	0	0	0	0	0	0	0	

☺ 当 ADLR=1，A/D 转换结果左对齐时，ADC 结果的保存方式为

bit	15	14	13	12	11	10	9	8	
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
读/写	R	R	R	R	R	R	R	R	
初始值	0	0	0	0	0	0		0	
bit	7	6	5	4	3	2	1	0	
	ADC1	ADC0	—	—	—	—	—	—	ADCL
读/写	R	R	R	R	R	R	R	R	
初始值	0	0	0	0	0	0	0	0	

【特殊功能 I/O 寄存器 SFIOR】

bit	7	6	5	4	3	2	1	0
	ADTS2	ADTS1	ADTS0	—	ACME	PUD	PSR2	PSR10
读/写	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ ADTS2 ~ ADTS0（bit7 ~ bit5）：ADC 自动转换触发源选择位。当 ADCSRA 寄存器中的 ADATE 为“1”，允许 ADC 工作在自动转换触发工作模式时，这 3 位的设置用于选择 ADC 的自动转换触发源，见表 11-6。如果禁止了 ADC 的自动转换触发（ADATE 为“0”），这 3 个位的设置值将不起任何作用。

表 11-6　ADC 自动转换触发源的选择

ADTS2	ADTS1	ADTS0	触　发　源	ADTS2	ADTS1	ADTS0	触　发　源
0	0	0	连续自由转换	1	0	0	T/C0 溢出
0	0	1	模拟比较器	1	0	1	T/C1 比较匹配 B
0	1	0	外部中断 0	1	1	0	T/C1 溢出
0	1	1	T/C0 比较匹配	1	1	1	T/C1 输入捕捉

3）ADC 应用设计要点

【预分频与转换时间】在通常情况下，ADC 的逐次比较转换电路需要一个 50~200kHz 之间的采样时钟才能达到最大精度。如果转换精度低于 10 位，则 ADC 的采样时钟可以高于 200kHz，以获得更高的采样率。

ADC 模块中包含一个带预分频器的 ADC 时钟源，如图 11-3 所示。预分频器可以对大于 100kHz 的系统时钟进行分频，以获得合适的 ADC 时钟提供 ADC 使用。预分频器的分频系数由 ADCSRA 寄存器中的 ADPS 位进行设置。一旦寄存器 ADCSRA 中的 ADEN 位置“1”（ADC 开始工作），预分频器就启动开始计数。只要 ADEN 位为“1”，预分频器将一直

工作，直到 ADEN 位为“0”。

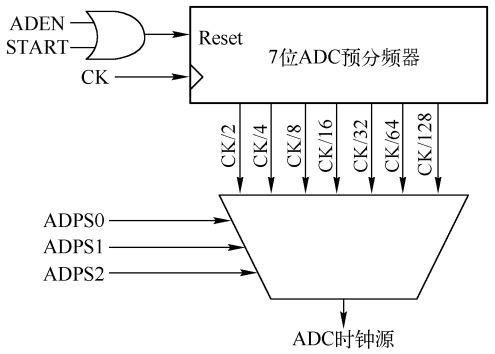


图 11-3　带预分频器的 ADC 时钟源

ADCSRA 寄存器中的 ADSC 置“1”后，在下一个 ADC 时钟周期的上升沿开始启动 ADC 进行 A/D 转换。AVR 的 ADC 完成一次转换的时间见表 11-7。正常转换需要 13～14 个 ADC 时钟。而启动 ADC 开始第 1 次转换到完成的时间需要 25 个 ADC 时钟，这是因为要对 ADC 单元的模拟电路部分进行初始化。

表 11-7　ADC 转换和采样保持时间

转 换 形 式	采样保持时间	完成转 换 总 时 间
启动 ADC 后的第一次转换	13.5 个 ADC 时钟	25 个 ADC 时钟
正常转换，单端输入	1.5 个 ADC 时钟	13 个 ADC 时钟
自动触发方式	2 个 ADC 时钟	13.5 个 ADC 时钟
正常转换，差分输入	1.5/2.5 个 ADC 时钟	13/14 个 ADC 时钟

在普通的 A/D 转换过程中，采样保持在转换启动后的 1.5 个 ADC 时钟开始；而第 1 次 A/D 转换的采样保持则发生在转换启动后的 13.5 个 ADC 时钟。当一次 A/D 转换完成后，转换结果写入 ADC 数据寄存器，ADIF（ADC 中断标志位）将被置位。在单次转换模式下，ADSC 也同时被清零。之后，用户可以将 ADSC 位写“1”，从而在 ADC 的第 1 个上升沿启动一次新的转换。

当 ADC 设置为自动触发方式时，触发信号的上升沿将启动 1 次 A/D 转换。这就保证了使 ADC 每隔一定的时间间隔进行一次转换。在这种方式下，ADC 需要 2 个 ADC 时钟周期的采样保持时间。为了实现同步逻辑，需要额外的 3 个 CPU 时钟周期。如果使用差分模式，加上不是由 A/D 转换结束实现的自动触发，每次转换需要 25 个 ADC 时钟周期。因为每次转换结束后都要关闭 ADC，然后又启动它。

在自由连续转换模式下，一次转换完毕后马上开始一次新的转换，此时，ADSC 位一直保持为“1”。

【ADC 输入通道和参考电源的选择】寄存器 ADMUX 中的 MUX[4..0]和 REFS1、REFS0 位实际上构成了一个缓冲器，该缓冲器与一个 MCU 可以随机读取的临时寄存器相连通。采用这种结构，保证了 ADC 输入通道和参考电源只能在 A/D 转换过程中的安全点被改变。在 A/D 转换启动前，通道和参考电源的选择可随时更改。但是，一旦启动 ADC 进行转换，通道和参考电源将被锁定，不允许再进行更改了，从而保证 ADC 有足够的采样时间，以确保

ADC 转换的正常进行。在转换完成前的最后一个 ADC 时钟周期（ADCSRA 的 ADIF 位置“1”时），通道和参考电源的选择又可以重新开始更新。

📖 注意:

由于 A/D 转换开始于置位 ADSC 后的第 1 个 ADC 时钟的上升沿，因此用户在置位 ADSC 后的一个 ADC 时钟周期内，不要将一个新的通道或参考电源写入到 ADMUX 寄存器中。

当改变差分输入通道时，一旦选定了差分输入通道，增益放大器就要用 125μs 的时间来稳定该值。所以在选择了新的差分输入通道后的 125μs 内不要启动 A/D 转换，或者将这段时间内的转换结果丢弃。

当改变 ADC 参考值后（通过改变 ADMUX 中的 REFS1、REFS0 位）的第 1 次转换也要遵循以上的时间处理过程。

选择模拟 C 输入通道时，应注意以下事项。

☺ 在单次转换模式下，总是在启动转换前选定通道。在 ADSC 置位后的一个 ADC 时钟周期就可以选择新的模拟输入通道了。但是最简单的办法是等待转换结束后再改变通道。

☺ 在连续转换模式下，总是在启动 ADC 开始第一次转换前改变通道设置。在 ADSC 置位后的一个 ADC 时钟周期就可以选择新的模拟输入通道了。但是最简单的方法是等到第一次转换完成后再改变通道的设置。然而，由于此时新一次的转换已经自动开始，所以，当前这次的转换结果仍反映前一通道的转换值，而下一次的转换结果将为新设置通道的值。

☺ 当切换到差分增益通道，由于自动偏移抵消电路需要沉积时间，第 1 次转换结果准确率很低，因此用户最好舍弃第 1 次转换结果。

ADC 的参考电压源（V_{REF}）决定了 ADC 转换的范围。如果单端通道的输入电压超过 V_{REF}，将导致转换结果接近于 0x3FF。ADC 的参考电压 V_{REF} 可以选择为 AV_{CC} 或芯片内部的 2.56V 参考源，或者为外接于 AREF 引脚上的参考电压源。

AV_{CC} 通过一个无源开关连接到 ADC。ADC 内部的 2.56V 参考源是由内部能隙参考源（V_{BC}）通过内部的放大器产生的。无论哪种情况，外部 AREF 引脚都是直接与 ADC 相连的，因此可以通过外部在 AREF 引脚和地之间并接一个电容，以提高参考电压的抗噪性。可以通过使用高阻电压表测量 AREF 引脚，来获得参考电源 V_{REF} 的电压值。由于 V_{REF} 是一个高阻源，因此，只有容性负载可以连接到该引脚。

如果将一个外部固定的电压源连接到 AREF 引脚，那么用户就不能选择其他的内部参考电源，否则就会导致片内基准源与外部参考源的短路。如果 AREF 引脚没有连接任何外部参考源，用户可以选择 AVcc 或 1.1V 作为基准源。参考电源改变后的第一次 A/D 转换结果可能不太准确，建议用户不要使用这一次的转换结果。

【ADC 转换结果】ADC 转换结束后（ADIF = 1），转换结果被存入 ADC 数据寄存器（ADCL 和 ADCH）。

☺ 对于单端输入的 ADC 转换，其转换结果为

$$ADC = (V_{IN} \times 1024) / V_{REF}$$

式中，V_{IN} 表示选定的输入引脚上的电压，V_{REF} 表示选定的参考电源的电压。如果转换

结果为 0x000，表示输入引脚的电压为模拟地；转换结果为 0x3FF，表示输入引脚的电压为参考电压值减去 1LSB。

☺ 对于差分转换，其结果为

$$ADC=(V_{POS}-V_{NEG})\times GAIN\times 512/V_{REF}$$

式中，V_{POS} 为输入引脚正电压，V_{NEG} 为输入引脚负电压，GAIN 为选定的增益系数，V_{REF} 表示选定的参考电源的电压。结果用 2 的补码形式表示，数值范围为 0x200～0x1FF。若用户希望对结果执行快速极性检测，应该读取转换结果的 MSB 位。如果该位为 1，表示结果为负；该位为 0，表示结果为正。

例如，若差分输入通道选择为 ADC3～ADC2，10 倍增益，参考电压 2.56V，左端对齐（ADMUX=0xED），ADC3 引脚上电压 300mV，ADC2 引脚上电压 500mV。

则 ADCR =（300−500）×10×512 / 2560 = −400 = 0x270

若结果为右端对齐时（ADLAR=0），则 ADCL=0x70，ADCH=0x02。

任务 2　系统电源电压监测器的设计

在一些使用干电池供电的便携式和手持式系统中，系统需要对电源电压进行监测，一旦电压低于某个值时，就要给出警告，提示用户更换电池，或者对电池进行充电。在此，使用 ATmega16 内置模拟比较器，设计一个单片机系统电源电压检测器。比较 PA0 口的电压与单片机内部的固定能隙电压（1.22V）之间的高低，如果 PA0 口的电压低于 1.22V，红色 LED 点亮，并发出报警声音，否则绿色 LED 点亮。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 11-8 所列的元器件，并完成图 11-4 所示的硬件电路图设计。

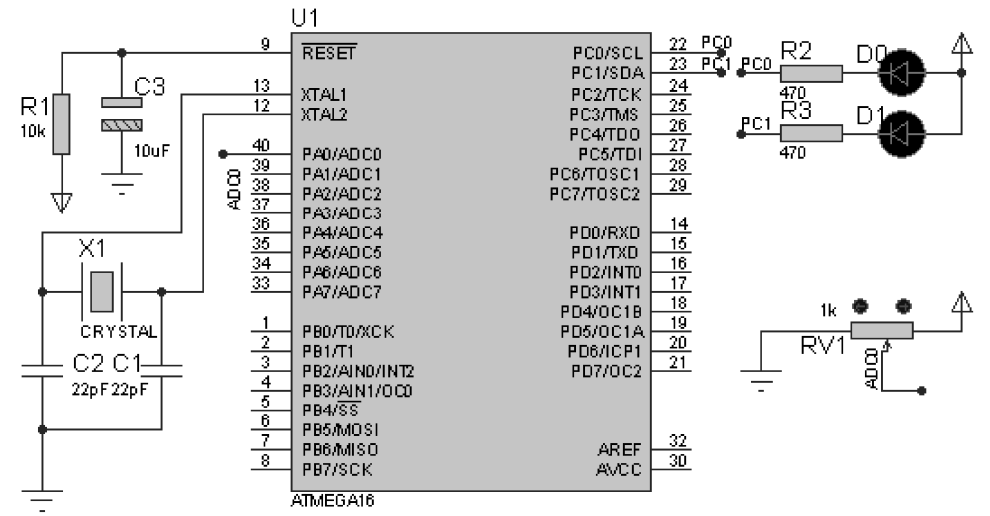


图 11-4　系统电源电压监测器原理图

表 11-8　系统电源电压监测器所用元器件

单片机 ATmega16	晶振 CRYSTAL 8MHz	电阻 RES	发光二极管 LED-RED
按钮 BUTTON	瓷片电容 CAP 22pF	可调电阻 POT-LOG	发光二极管 LED-GREEN

软件设计

在本设计中利用电位器（即可调电阻）组成分压电路，单片机的 PA0 口连接在电位器的动片引脚上，这个电路实际上相当于两个电阻组成的串联电路，只不过这两个电阻的电阻值是可变的，改变动片的位置，就可以改变 PA0(ADC0)与地之间的电阻值，根据串联电路分压的原理，当动片移动时，PA0 处的电压就会发生变化。设置 SFIOR 的 ACME 位为 1、ACSR 为 0x60，可以将 PA0 处的电压与内部的 1.22V 电压进行比较。如果 PA0 处的电压高于 1.22V 时，ACSR 寄存器中的 ACO 为 1，否则为 0。因此在编写程序时，可以先判断 ACSR 的 ACO 位是否为“0”，然后根据判断结果进行相应操作。如果该位为“0”，则点亮红色 LED；否则点亮绿色 LED。其程序流程图如图 11-5 所示。

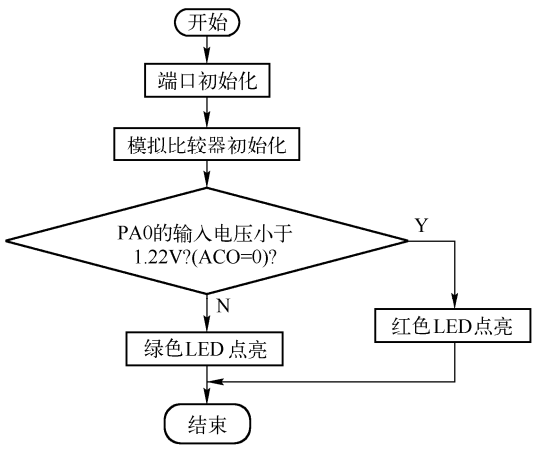


图 11-5　系统电源电压监测器程序流程图

源程序

```
/*****************************************************
  文件名:          系统电源电压检测器.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define   D0              PORTC.0
#define   D1              PORTC.1
void volt_monitor(void)           //电压监测
```

```
{
  if(ACSR.5= =1)                    //判断 ADC0 的电压
    {
      D0=0;                         //低于 1.22V，D0 点亮
      D1=1;
    }
  else
    {
      D0=1;
      D1=0;                         //高于 1.22V，D1 点亮
    }
}
void port_init(void)                //端口初始化
{
   DDRC=0xFF;
  PORTC=0xFF;
}
void comp_init(void){
   SFIOR=0x08;                      //比较器初始化作为模拟比较器反向输入端的信号源
                                    //同时选择 ADC 的 ADC0 作为输入端，并且关闭 AD 转换的
                                    //使能
   ACSR=0x60;                       //允许模拟比较器，AIN0 设置为内部固定能隙参考电源 1.22V
}
void main(void)
{
  port_init();
  comp_init();
  while(1)
    {
      volt_monitor();
    }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“系统电源电压监测器.COFF”文件，然后在 Proteus ISIS 中打开已创建的系统电源电压监测器电路图并进行相应设置。为了观察 ADC0(PA0)端的电压值，应在 Proteus 中添加电压探针。其操作步骤是，首先在 Proteus ISIS 编辑界面的工具箱中单击图标，然后在 ADC0 端单击鼠标左键，即可添加电压探针。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。滑动可调电阻，可以观看两个 LED 的显示状态及 ADC0 端输入的电压值。如果 PA0 端小于 1.22V，D0 点亮，否则 D1 点亮，其仿真效果如图 11-6 所示。

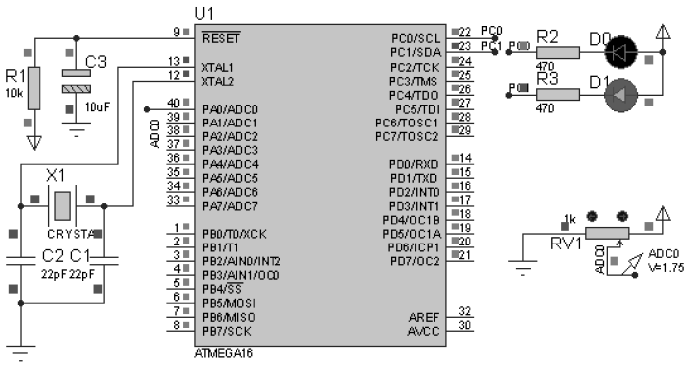


图 11-6　系统电源电压监测器仿真效果图

任务 3　低压报警器的设计

使用 ATmega16 监测两路外部输入电压情况，当 PB2 脚电压值小于 PB3 脚电压值时红色 LED 点亮，并发出报警声音，否则绿色 LED 点亮。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 11-9 所列的元器件，并完成图 11-7 所示的硬件电路图设计。

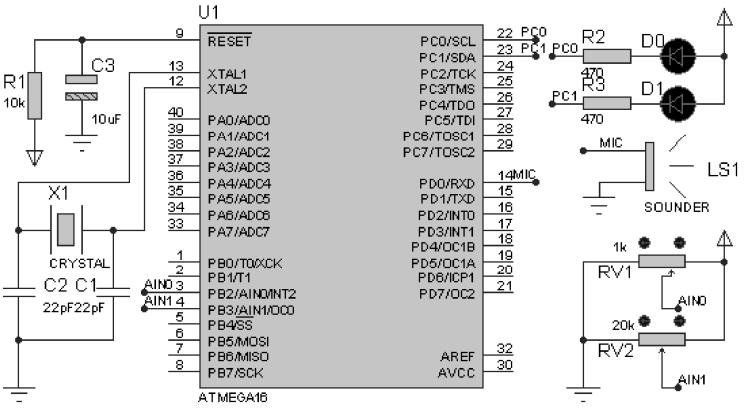


图 11-7　低压报警器电路图

表 11-9　低压报警器所用元器件

单片机 ATmega16	晶振 CRYSTAL 8MHz	按钮 BUTTON	电阻 RES
瓷片电容 CAP 22pF	发光二极管 LED-GREEN	蜂鸣器 SOUNDER	可调电阻 POT-LOG
发光二极管 LED-RED			

软件设计

ATmega16 的模拟比较器可以实现对正极 AIN0 和负极 AIN1 这两个输入端的模拟输入

电压进行比较。当 AIN0 上的电压高于 AIN1 的电压时，模拟比较器输出 ACO 置“1”；当 AIN0 上的电压低于 AIN1 的电压时，模拟比较器输出 ACO 清零。因此在编写程序时，可以先判断 ACSR 的 ACO 位是否为“0”，然后根据判断结果进行相应操作。如果该位为“0”，则点亮红色 LED，并发出报警声；否则点亮绿色 LED，程序流程如图 11-8 所示。

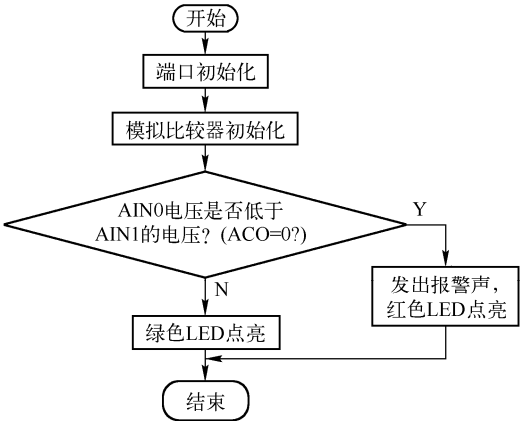


图 11-8　低电报警器程序流程图

源程序

```
/*****************************************************
  文件名:          低压报警器.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define   MIC              PORTD.0
#define   D0               PORTC.0
#define   D1               PORTC.1
bit     sp;
void delay(uint k)
{
   uint m,n;
   for(m=0;m<k;m++)
      {
        for(n=0;n<575;n++)
          {;}
      }
}
void sound(void)                    //声音报警
{
   uchar i;
   for(i=0;i<200;i++)
      {
          sp=~sp;
          MIC=sp;                   //输出频率 1kHz
          delay(1);                 //延时 500us
      }
   for(i=0;i<200;i++)
```

```
        {
            sp=～sp;
            MIC=sp;                    //输出频率 500Hz
            delay(2);                  //延时 1ms
        }
}
void volt_alarm(void)                  //低电压报警
{
  if(ACSR.5= =0)
        {
            D0=0;                      //AIN1>AIN0，低电压声光报警
            D1=1;
            sound();
        }
  else
        {
            D0=1;                      //AIN1<AIN0，电压正常
            D1=0;
        }
}
void port_init(void)                   //端口初始化
{
   DDRC=0xFF;
  PORTC=0xFF;
   DDRD=0xFF;
  PORTD=0xFF;
}
void comp_init(void)                   //比较器初始化
{
    ACSR=0x00;                         //ACIS1=0，ACIS0=0 比较器输出变化即触发中断
    SFIOR=0x80;
}
void main(void)
{
  port_init();
  comp_init();
  while(1)
        {
          volt_alarm();
        }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“低电报警器.COFF”文件，然后在 Proteus ISIS 中打开已创建的低电报警器电路图并进行相应设置。为了观察 AIN0 和 AIN1 端的电压值，应在 Proteus 中添加电压探针。其操作步骤是，首先在 Proteus ISIS 编辑界面的工具箱中单击图标，然后分别在 AIN0 和 AIN1 端单击鼠标左键，即可添加电压探针。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。滑动可调电阻，可以观看两个 LED 的显示状态及 AIN0 和 AIN1 端输入的电压值。如果 AIN1>AIN0 时，D0 点亮并发出报警声，否则 D1 点亮，其仿真效果如图 11-9 所示。

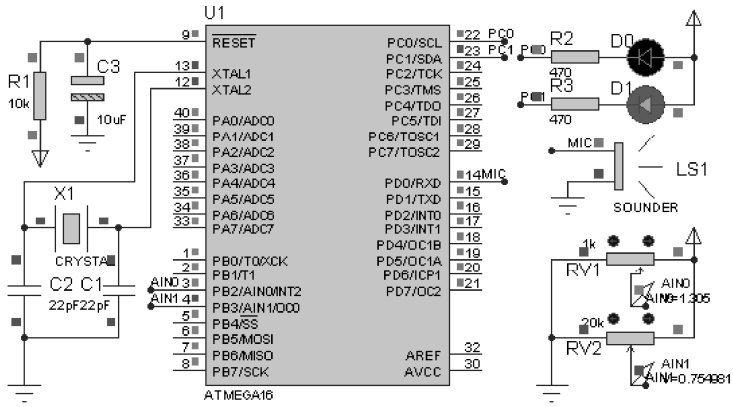


图 11-9　低电报警器仿真效果图

任务 4　简易电压表的设计

利用 ATmega16 的 A/D 转换器，设计一个范围为 0～5V 的简单数字电压表。要求使用电位器 RV1 产生电阻分压电路，从而产生变化的模拟电压加到单片机的模拟信号采集端口，供单片机采集；4 位共阳极能够显示所采集到的电压值。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 11-10 所列的元器件，并完成图 11-10 所示的硬件电路图设计。

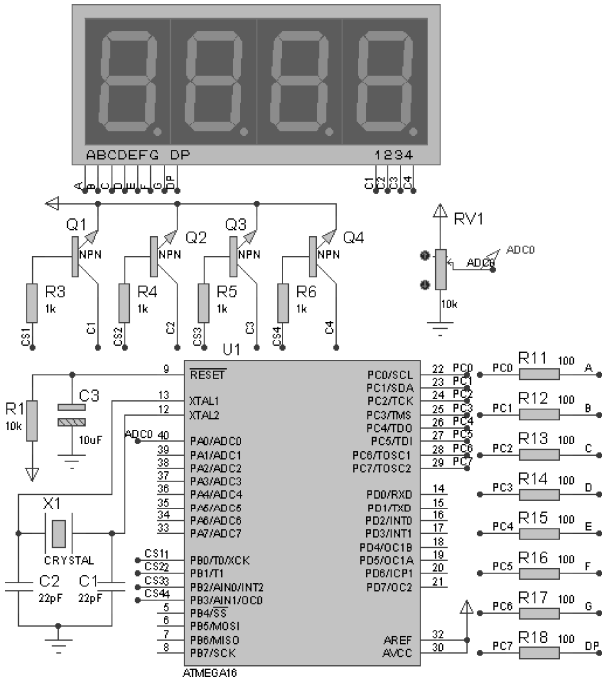


图 11-10　简易电压表电路图

表 11-10　简易电压表所用元器件

单片机 ATmega16	晶振 CRYSTAL 8MHz	按钮 BUTTON	电阻 RES
瓷片电容 CAP 22pF	可调电阻 POT-LOG	4 位共阳极 LED 数码管 7SEG-MPX4-CA-BLUE	

软件设计

在通常情况下，ADC 的逐次比较转换电路要达到最大精度时，需要 50～200kHz 之间的采样时钟。系统频率为 8MHz，它远大于 ADC 的采样时钟频率（ADCclk），因此需将此频率变成 ADC 所需的采样时钟频率。由于 ADC 模块中包含一个预分频器的 ADC 时钟源，它可以对大于 100kHz 的系统时钟进行分频，以获得合适的 ADC 时钟提供 ADC 使用。预分频器的分频系数由 ADCSRA 寄存器中的 ADPS[2..0]设置的。如果设置 ADPS[2..0]=110 时（即分频系数为 64），ADCclk=8MHz/64=125kHz，这样就可满足采样时钟的要求。

对于单端输入而言，正常转换需要 13～14 个 ADC 时钟，即启动下次转换时，必须保证上次转换时间至少为 14/125kHz=0.112ms，因此可以在程序中编写大于 0.112ms 的延时程序。在此可以采用 T/C0 的比较中断，每 2ms 中断一次，由于 2ms 远大于 0.112ms，所以它可以作为 ADC 自动触发转换的触发源信号。

在 A/D 转换完成中断服务中，需要将转换的结果换算成电压值。由于转换的结果为无符号整数型，在换算时，为了保证计算不会产生溢出，最好先将该数值的类型强制为长整型，再乘以 5000（因为 AVcc 的参考的电压为 5V），最后除以 1024，即可得到换算后的电压数据。

综上所述，该程序主要由 T/C0 比较匹配中断函数、A/D 转换完成中断函数、LED 显示驱动函数等部分组成。其程序流程图如图 11-11 所示。

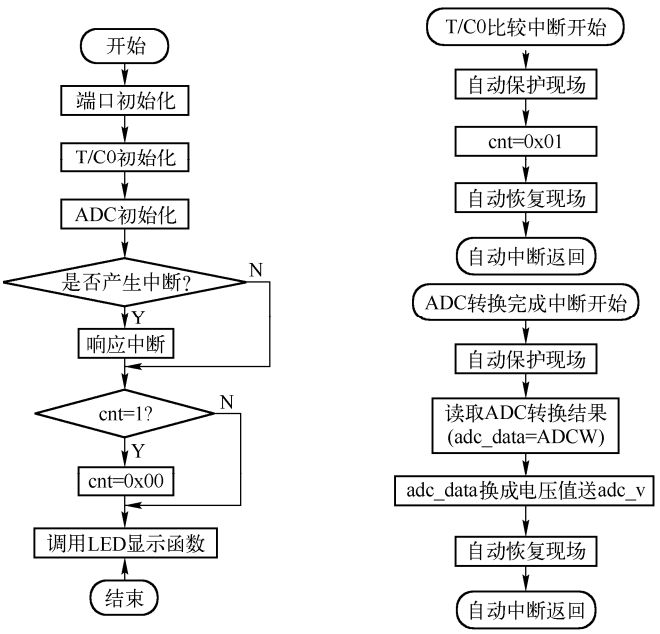


图 11-11　简易电压表程序流程图

源程序

```
/*****************************************************
 文件名:          简易电压表.c
 单片机型号:      ATmega16
 晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define ulong unsigned long
#define   CS1        PORTB.0
#define   CS2        PORTB.1
#define   CS3        PORTB.2
#define   CS4        PORTB.3
#define   LED         PORTC
flash uchar tab[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,   //共阳极 LED0～F 的段码
                   0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E,0xBF,0x10};
uchar cnt;
uint adc_v,val1,val2,val3,val4;
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<114;n++)
        {;}
    }
}
interrupt [TIM0_COMP] void timer0_comp_isr(void)      // T/C0 比较匹配中断服务
{
   cnt=0x01;
}
interrupt [ADC_INT] void adc_isr(void)                // ADC 转换完成中断服务
{
     uint adc_data;
     adc_data=ADCW;                                   //读取 ADC 转换结果
     adc_v=(ulong)adc_data*5000/1024;                 //换算成电压值
}
void volt_in(void)                                    //电压值各位分离
{
  val1=adc_v/1000;
  val2=(adc_v/100)%10;
  val3=(adc_v/10)%10;
  val4=adc_v%10;
}
void display(void)                                    //LED 数码管显示函数
 {
```

```
    PORTB=0x00;
    LED=tab[val1];
    PORTC.7=0;                          //小数点
    CS1=1;
    delay(1);
    CS1=0;
    LED=tab[val2];
    CS2=1;
    delay(1);
    CS2=0;
    LED=tab[val3];
    CS3=1;
    delay(1);
    CS3=0;
    LED=tab[val3];
    CS4=1;
    delay(1);
    CS4=0;
}
void port_init(void)
{
  DDRA=0x00;                            //模拟量由 PA0 输入
  PORTA=0x00;
  DDRB=0x0F;                            //LED 片选控制
  PORTB=0x00;
  DDRC=0xFF;                            //LED 段码输出
  PORTC=0x00;
}
void timer0_init(void)
{
  TCCR0=0x0B;                           //内部时钟，64 分频（8M/64=125kHz），
                                        //CTC 模式
  TCNT0=0x00;
  OCR0=0xF9;                            //OCR0 = 0xF9(即 249)，(249+1)/125=2ms
  TIMSK=0x02;                           //允许 T/C0 比较中断
}
void ADC_init(void)
{
  ADMUX=0x40;                           //参考电源 AVcc、ADC0 单端输入
  SFIOR&=0x1F;
  SFIOR|=0x60;                          //选择 T/C0 比较匹配中断为 ADC 触发源
  ADCSRA=0xAE;                          //ADC 允许、自动触发转换、ADC 转换
                                        //中断允许、ADCclk=8MHz/64=125kHz
  #asm("sei")                           //开放全局中断
}
void main(void)
{
```

```
    port_init();
    timer0_init();
    ADC_init();
    while(1)
      {
        if(cnt==1)
          {
            volt_in();
            display();
            cnt=0;
          }
      }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“简易电压表.COFF”文件，然后在 Proteus ISIS 中打开已创建的简易电压表电路图并进行相应设置。为了观察 ADC0 端的理论电压值，应在 Proteus 中添加电压探针。单击 Proteus ISIS 模拟调试按钮中的运行按钮 ，进入调试状态。滑动可调电阻，可以观看 4 位共阳极数码管显示的电压值也会有所变化，如图 11-12 所示。多次滑动可调电阻，可以看出，LED 数码管显示的电压值与电压探针显示的电压值略有不同，只不过两个值的误差较小，用户可以认为 LED 数码管显示的电压值即为实际所测电压值。

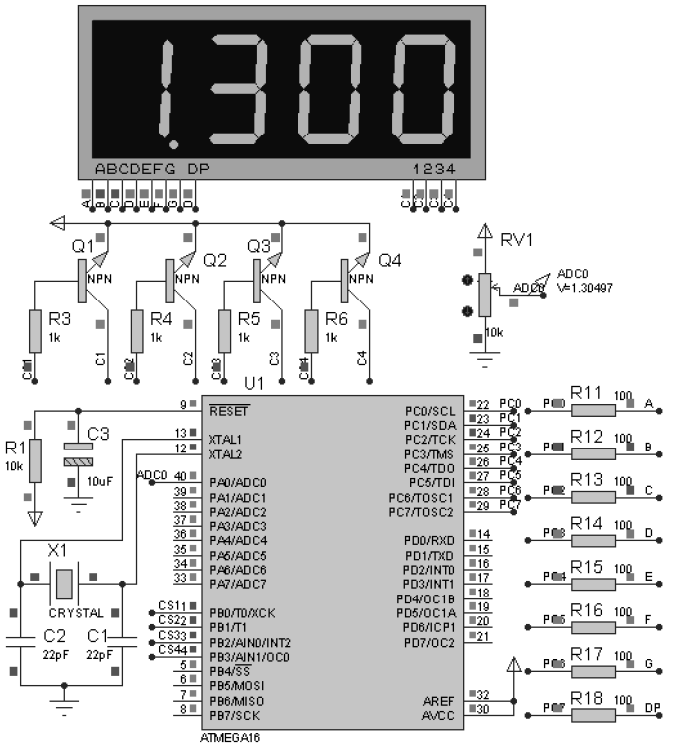


图 11-12　简易电压表仿真效果图

任务 5　电压监控器的设计

在自动化控制中，有些设备需要在一定的电压下工作，在此采用内置的 A/D 转换器设计一个电压监控器，并使用 3 种不同颜色的 LED 来模拟设备的工作运行状态。当工作电压小于 2.000V 时，黄色 LED 点亮，表示设备处于欠电压运行状态；当工作电压大于 3.000V 时，红色 LED 点亮，表示设备处于过电压运行状态；当工作电压在 2.000～3.000V 时，绿色 LED 点亮，表示设备处于正常工作电压运行状态。所测电压值及状态由 LCD1602 进行显示。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 11-11 所列的元器件，并完成图 11-13 所示的硬件电路图设计。

表 11-11　电压监控器所用元器件

单片机 ATmega16	晶振 CRYSTAL 8MHz	按钮 BUTTON	电阻 RES
瓷片电容 CAP 22pF	发光二极管 LED-GREEN	蜂鸣器 SOUNDER	可调电阻 POT-LOG
发光二极管 LED-RED	发光二极管 LED-BULE	LM016L 字符式 LCD	

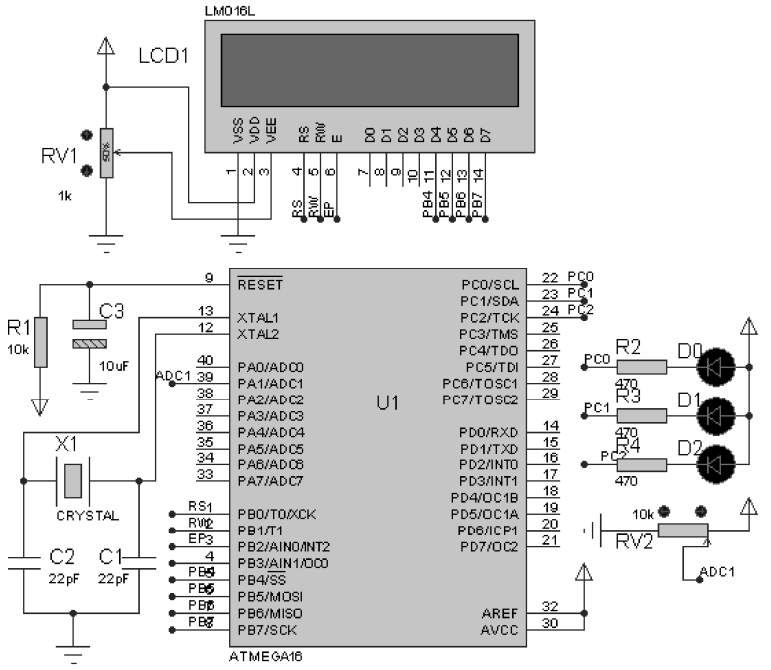


图 11-13　电压监控器电路图

软件设计

本设计是在任务 4 的基础上进行的，首先将 ADC1 中的 A/D 转换为相应的数字量，再

将该数据进行比较处理及 LCD 显示字符转换处理，然后 LCD 进行相应的显示控制即可。由于是对 ADC1 输入的模拟值进行转换，因此 ADC 初始化时，ADMUX 值应为 0x41。其主要程序流程图与图 11-11 类似，在此省略本设计的程序流程图。

源程序

```
/*****************************************************
  文件名:          电压监控器.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#include   "lcd.h"
#define uchar unsigned char
#define uint unsigned int
#define ulong unsigned long
#define   D0              PORTC.0
#define   D1              PORTC.1
#define   D2              PORTC.2
#asm
.equ __lcd_port=0x18;                                  //PORTB 数据寄存器地址
#endasm
typedef uchar byte;
flash char dis_str1[]="   Voltage < 2V   ";
flash char dis_str2[]="2V< Voltage < 3V";
flash char dis_str3[]="   Voltage > 3V   ";
uchar lcdd[5];
char flash *str;
uchar cnt;
uint adc_v;
uchar val1,val2,val3,val4;
/*void   delay(uint k)                                 //延时约 0.1ms
{
  uchar   m,n;
   for(m=0;m<k;m++)
    {
     for(n=0;n<114;n++);
    }
}*/
interrupt [TIM0_COMP] void timer0_comp_isr(void)       //T/C0 比较匹配中断服务
{
   cnt=0x01;
}
interrupt [ADC_INT] void adc_isr(void)                 //ADC 转换完成中断服务
{
    uint adc_data;
    adc_data=ADCW;                                     //读取 ADC 置换结果
    adc_v=(ulong)adc_data*5000/1024;                   //换算成电压值
```

```
}
void volt_in(void)                                  //电压值各位分离
{
   val1=adc_v/1000;
   val2=(adc_v/100)%10;
   val3=(adc_v/10)%10;
   val4=adc_v%10;
}
void volt_comp(void)
{
   if(adc_v<=2000)                                  //低电压
       {
          str=dis_str1;
          D0=0;                                     //红灯亮
          D1=1;
          D2=1;
       }
    else if(adc_v>3000)
       {
          str=dis_str3;
          D0=1;
          D1=1;
          D2=0;                                     //蓝灯亮
       }
    else
       {
          str=dis_str2;
          D0=1;
          D1=0;                                     //绿灯亮
          D2=1;
       }
}
void lcd_in(void)                                   //LCD 电压值转换为相应 ASCII 码
{
   volt_in();
   lcdd[3]=val4+0x30;
   lcdd[2]=val3+0x30;
   lcdd[1]=val2+0x30;
   lcdd[0]=val1+0x30;
}

void display0(void)                                 //LCD 第 1 行显示内容
{
    lcd_in();
    lcd_gotoxy(0,0);
    lcd_putsf("Voltage:");
    lcd_gotoxy(9,0);
    lcd_putchar(lcdd[0]);
    lcd_gotoxy(10,0);
```

```
    lcd_putsf(".");
    lcd_gotoxy(11,0);
    lcd_putchar(lcdd[1]);
    lcd_gotoxy(12,0);
    lcd_putchar(lcdd[2]);
    lcd_gotoxy(13,0);
    lcd_putchar(lcdd[3]);
    lcd_gotoxy(14,0);
    lcd_putchar(lcdd[4]);
    lcd_gotoxy(15,0);
    lcd_putsf("v");
}
void display1(void)                          //LCD 第 2 行显示内容
{
    lcd_gotoxy(0,1);
    lcd_putsf(str);
}
void display(void)                           //LCD 显示
{
  display0();
  display1();
}
void port_init(void)                         //端口初始化
{
     DDRA=0x00;
    PORTA=0x00;
     DDRC=0xFF;
    PORTC=0xFF;
}
void timer0_init(void)
{
    TCCR0=0x0B;                              //内部时钟，64 分频（8M/64=
                                             //125kHz），CTC 模式
    TCNT0=0x00;
    OCR0=0xF9;                               // OCR0 = 0xF9（249），（249+1）
                                             //125=2ms
    TIMSK=0x02;                              //允许 T/C0 比较中断
}
void ADC_init(void)
 {
    ADMUX=0x41;                              //参考电源 AVcc、ADC1 单端输入
    SFIOR&=0x1F;
    SFIOR|=0x60;                             //选择 T/C0 比较匹配中断为 ADC
                                             //触发源
    ADCSRA=0xAE;                             // ADC 允许、自动触发转换、ADC
                                             //转换中断允许、
                                             //ADCclk=8MHz/64=125kHz
    #asm("sei")                              //开放全局中断
 }
void   main(void)
{
  port_init();
```

```
    timer0_init();
    ADC_init();
    lcd_init(16);
    while(1)
      {
        if(cnt= =1)
         {
           cnt=0;
           volt_comp();
           display();
         }
      }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“电压监控器.COFF”文件，然后在 Proteus ISIS 中打开已创建的电压监控器电路图并进行相应设置。为了观察 ADC1 端的理论电压值，应在 Proteus 中添加电压探针。单击 Proteus ISIS 模拟调试按钮中的运行按钮 ，进入调试状态。滑动可调电阻，可以观看 LCD 显示的电压值也会有所变化。多次滑动可调电阻，可以看出，LCD 显示的电压值与电压探针显示的电压值略有不同，只不过两个值的误差较小，用户可以认为 LCD 显示的电压值即为实际所测电压值。当 LCD 显示的电压值小于 2V 时，红色 LED 将点亮；当 LCD 显示的电压值大于 3V 时，蓝色 LED 将点亮；当 LCD 显示的电压值大于 2V 且小于 3V 时，绿色 LED 将点亮，如图 11-14 所示。

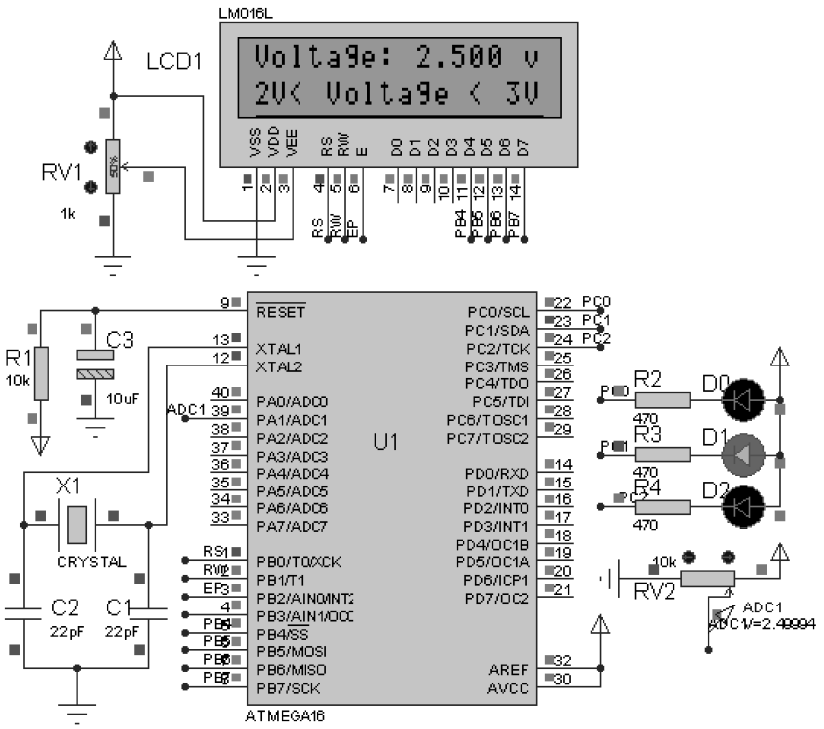


图 11-14　电压监控器仿真效果图

项目十二　SPI 串行总线的应用

〖知识目标〗

☺ 了解 ATmega16 内部 SPI 总线接口结构、工作模式、相关的控制寄存器定义;

☺ 了解 ATmega16 内部 SPI 总线在单片机应用系统中的使用方法。

〖能力目标〗

☺ 能正确使用 ATmega16 内部 SPI 总线与外部器件的连接;

☺ 掌握单片机对 SPI 串行总线器件的控制与编程。

SPI（Serial Peripheral Interface）总线是由 Freescale 公司（原 Motorola 公司半导体部）最先推出的一种串行总线技术，它是在芯片之间通过串行数据线（MISO、MOSI）和串行时钟线（SCLK）实现同步串行数据传输的技术。

任务 1　SPI 总线基础知识认知

1．SPI 总线接口的内部结构和功能

ATmega16 的 SPI 总线接口由数据寄存器、时钟逻辑、引脚逻辑和控制逻辑等部分组成，如图 12-1 所示。SPI 主机与从机的连接如图 12-2 所示。

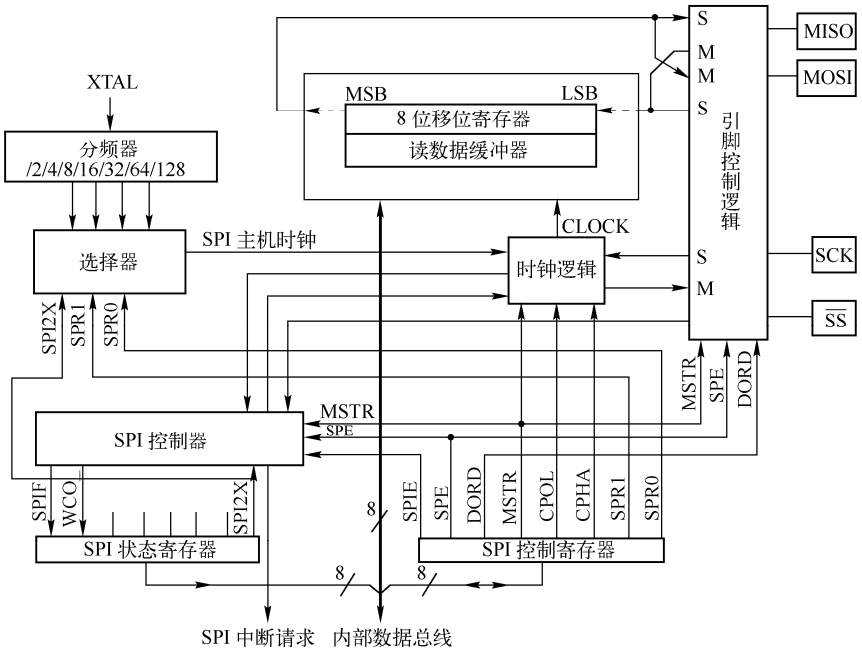


图 12-1　ATmega16 的 SPI 总线接口的内部结构

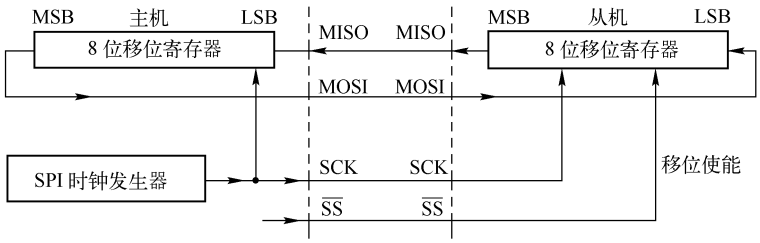


图 12-2　SPI 主机和从机的连接

从图 12-2 中可看出，系统包括两个移位寄存器和一个主机时钟发生器。通过将需要的从机的$\overline{SS}$引脚拉低，主机启动一次通信过程。主机和从机将需要发送的数据放入相应的移位寄存器。主机在 SCK 引脚上产生时钟脉冲以交换数据。主机的数据由主机的 MOSI 移位输出，从机通过从机的 MOSI 移位输入；从机的数据由从机的 MISO 移位输出，主机通过主机的 MISO 移位输入。主机通过将从机的$\overline{SS}$拉高实现与从机的同步。

将 ATmega16 配置为 SPI 主机时，SPI 接口不自动控制$\overline{SS}$引脚，必须由用户软件来处理。对 SPI 数据寄存器写入数据时启动 SPI 时钟，将 8 位的数据移入从机。传输结束后 SPI 时钟停止，传输结束标志 SPIF 置位。如果此时 SPCR 寄存器的 SPI 中断使能位 SPIE 置位，中断就会发生。主机可以继续向 SPDR 写入数据以移位到从机中去，或者是将从机的$\overline{SS}$拉高以说明数据包发送完成。最后进来的数据将一直保存于缓冲寄存器里。

将 ATmega16 配置为 SPI 从机时，只要$\overline{SS}$为高，SPI 接口将一直保持睡眠状态，并保持 MISO 为三态。在这种状态下，软件可以更新 SPI 数据寄存器 SPDR 的内容。即使此时 SCK 引脚有输入时钟，SPDR 的数据也不会移出，直至$\overline{SS}$被拉低。一个字节完全移出后，传输结束标志 SPIF 置位。如果此时 SPCR 寄存器的 SPI 中断使能位 SPIE 置位，就会产生中断请求。在读取移入的数据前，从机可以继续往 SPDR 写入数据。最后进来的数据将一直保存于缓冲寄存器中。

SPI 系统的发送端只有一个缓冲器，而在接收端有两个缓冲器。也就是说，在发送时一定要等到移位过程全部结束后才能对 SPI 数据寄存器执行写操作。而在接收数据时，需要在下一个字符移位过程结束前，通过访问 SPI 数据寄存器才能读取当前接收到的字符；否则第一个字节将丢失。

ATmega16 工作于 SPI 从机模式时，控制逻辑对 SCK 引脚的输入信号进行采样。为了保证对时钟信号的正确采样，SPI 时钟不能超过 $f_{osc}/4$。SPI 使能后，MOSI、MISO、SCK 和$\overline{SS}$引脚的数据方向将按照表 12-1 所列自动进行配置。

表 12-1　SPI 引脚方向定义

引　脚	SPI 主机	SPI 从机	引　脚	SPI 主机	SPI 从机
MOSI	用户定义	输入	SCK	用户定义	输入
MISO	输入	用户定义	$\overline{SS}$	用户定义	输入

2. $\overline{SS}$引脚功能

【从机模式】当 SPI 配置为从机时，从机选择$\overline{SS}$引脚总是为输入。$\overline{SS}$引脚为低电平时，将激活 SPI 接口，MISO 成为输出引脚，其他引脚成为输入引脚。当$\overline{SS}$引脚为高电平

时，所有的引脚成为输入，SPI 逻辑复位，不再接收数据。

$\overline{SS}$ 引脚对于数据包/字节的同步非常有用，可以使从机的位计数器与主机的时钟发生器同步。当 $\overline{SS}$ 引脚拉为高电平时，SPI 从机立即复位接收和发送逻辑，并丢弃移位寄存器里不完整的数据。

【主机模式】当 SPI 配置为主机时（MSTR 的 SPCR 置位），用户可以决定 $\overline{SS}$ 引脚的方向。若 $\overline{SS}$ 引脚配置为输出，则此引脚可以用做普通的 I/O 端口而不影响 SPI 系统；如果 $\overline{SS}$ 引脚配置为输入，$\overline{SS}$ 必须保持为高电平以保证 SPI 的正常工作。

若系统配置为主机，$\overline{SS}$ 引脚为输入，但被外设拉为低电平，则 SPI 系统会将此低电平解释为有一个外部主机将自己选择为从机。为了防止总线冲突，SPI 系统将实现如下动作。

☺ 清零 SPCR 的 MSTR 位，使 SPI 成为从机，从而 MOSI 和 SCK 变为输入。

☺ SPSR 的 SPIF 置位。若 SPI 中断和全局中断开放，则中断服务程序将得到执行。

因此，使用中断方式处理 SPI 主机的数据传输，并且存在 SS 被拉低的可能性时，中断服务程序应该检查 MSTR 是否为“1”。若被清零，用户必须将其置位，以重新使能 SPI 主机模式。

3．SPI 数据模式

为适应不同外部设备的串行通信，可通过程序来改变 SPI 串行时钟的相位和极性，即选择 CPOL 与 CPHA 的 4 种不同组合方式，见表 12-2。其中 CPOL 用于选择时钟极性，与发送格式无关。而时钟相位 CPHA 用于控制两种发送格式（CPHA=0 和 CPHA=1 的发送格式）。对于主、从机通信，时钟相位和极性必须相同。

表 12-2　CPHA 和 CPOL 控制组合

SPI 模式	CPHA	CPOL	起　始　沿	结　束　沿	参　考　图
0	0	0	采样（上升沿）	设置（下降沿）	图 12-3
1	1	0	设置（下降沿）	采样（上升沿）	图 12-3
2	0	1	采样（上升沿）	设置（下降沿）	图 12-4
3	1	1	设置（下降沿）	采样（上升沿）	图 12-4

【CPHA=0 的发送格式】图 12-3 所示为 CPHA=0 的发送格式。图中 SCK 有两种波形，一种为 CPOL=0，另一种为 CPOL=1。在 CPHA=0 时，$\overline{SS}$ 下降沿用于启动从机数据发送，而第一个 SCK 跳变捕捉最高位。在一次 SPI 传送完毕，从机的 $\overline{SS}$ 脚必须返回高电平。

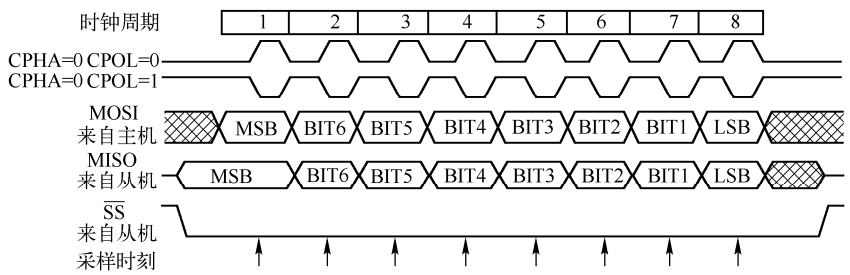


图 12-3　CPHA 为 0 时传送格式

【CPHA=1 的发送格式】图 12-4 所示为 CPHA=1 的发送格式。这时，主机在 SCK 的第

一个跳变后开始驱动 MOSI，从机应用它来启动数据发送。SPI 传送期间，从机的 $\overline{SS}$ 引脚保持为低电平。

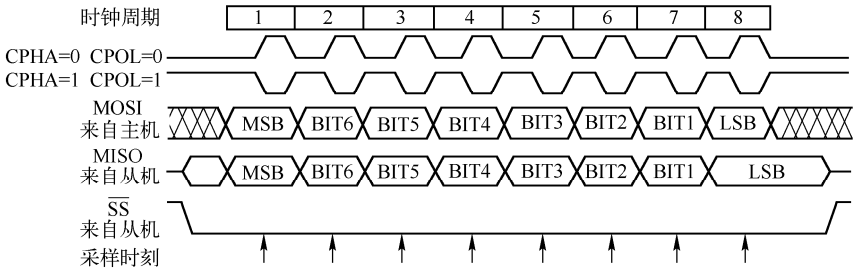


图 12-4　CPHA 为 1 时传送格式

4．SPI 相关寄存器

【SPI 控制寄存器 SPCR】

Bit	7	6	5	4	3	2	1	0
	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
读 / 写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ SPIE（bit7）：使能 SPI 中断控制位。SPIE=1 时，只要 SPSR 寄存器的 SPIF 和 SREG 寄存器的全局中断使能位置 1，则系统响应 SPI 中断。

☺ SPE（bit6）：使能 SPI 控制位。SPE=1 时，允许 SPI 接口。进行任何 SPI 操作前，应先将该位置 1。

☺ DORD（bit5）：数据移出顺序控制位。DORD=1 时，数据首先传送 LSB 位，即低位在前；DORD=0 时，数据首先传送 MSB 位，即高位在前。

☺ MSTR（bit4）：主/从机选择位。MSTR=1 时，选择 SPI 为主机方式；MSTR=0 时，选择 SPI 为从机方式。如果 MSTR 为 1，$\overline{SS}$ 配置为输入，但被拉低，则 MSTR 被清零，寄存器 SPSR 的 SPIF 置位。用户必须重新设置，才能再次将 SPI 设置为主机方式。

☺ CPOL（bit3）：时钟极性选择位。CPOL=1，表示空闲时 SCK 为高电平；CPOL=0，表示空闲时 SCK 为低电平。

☺ CPHA（bit2）：时钟相位选择位。CPHA 决定数据是在 SCK 的起始沿采样还是在 SCK 的结束沿采样。CPOL 和 CPHA 决定了 SPI 的工作模式，请参见表 12-2。

☺ SPR1、SPR0（bit1、bit0）：SPI 时钟速率选择位。这两个标志位与寄存器 SPSR 中的 SPI2X 位一起，用于设置主机模式下产生的串行时钟 SCK 速率。SPR1 和 SPR0 对于从机模式无影响，SCK 和振荡器频率 f_{osc} 之间的关系见表 12-3。

表 12-3　SCK 和振荡器频率的关系

SPI2X	SPR1	SPR0	SCK 频率	SPI2X	SPR1	SPR0	SCK 频率
0	0	0	$f_{osc}/4$	1	0	0	$f_{osc}/2$
0	0	1	$f_{osc}/16$	1	0	1	$f_{osc}/8$
0	1	0	$f_{osc}/64$	1	1	0	$f_{osc}/32$
0	1	1	$f_{osc}/128$	1	1	1	$f_{osc}/64$

【SPI 状态寄存器 SPSR】 SPI 状态寄存器 SPSR 在 I/O 空间的地址为 0x0E(0x002E)，与 SPI 有关的只有 3 个标志位，各位定义如下：

Bit	7	6	5	4	3	2	1	0
	SPIF	WCOL	—	—	—	—	—	SPI2X
读 / 写	R	R	R	R	R	R	R	R/W
初始值	0	0	0	0	0	0	0	0

☺ SPIF（bit7）：SPI 中断标志位。当串行传送完成时，SPIF 置 1。如果此时寄存器 SPCR 的 SPIE 和全局中断使能位置 1，SPI 中断即产生。如果 SPI 为主机，$\overline{SS}$ 配置为输入，且被拉低，SPIF 也将置 1。进入中断服务程序后，SPIF 自动清零。或者可以通过先读 SPSR，紧接着访问 SPDR 来对 SPIF 清零。

☺ WCOL（bit6）：写冲突标志。如果在发送过程中，对 SPI 数据寄存器 SPDR 写数据时，WCOL 将置 1。WCOL 可以通过先读 SPSR，紧接着访问 SPDR 来清零。

☺ SPI2X（bit0）：SPI 倍速选择位。在主机方式下，当该位写为逻辑“1”时，SPI 的速度（SCK 的频率）将加倍，见表 12-3。在从机方式下，SCK 必须低于 $f_{osc}/4$，才能确保有效的数据传送。

【SPI 数据寄存器 SPDR】SPI 数据寄存器 SPDR 为读/写寄存器，用于在寄存器文件的 SPI 移位寄存器之间传输数据，其 I/O 空间的地址为 0x0F(0x002F)。写寄存器将启动数据传输，读寄存器将读取寄存器的接收缓冲器，各位定义如下：

Bit	7	6	5	4	3	2	1	0
	MSB							LSB
读 / 写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	×	×	×	×	×	×	×	×

任务 2　74LS164 驱动数码管显示可调数字钟

使用 74LS164 外接 8 位共阳数码管，设计一个可调数字钟，要求 74LS164 由 Atmega 16 的内置 SPI 总线接口进行驱动。

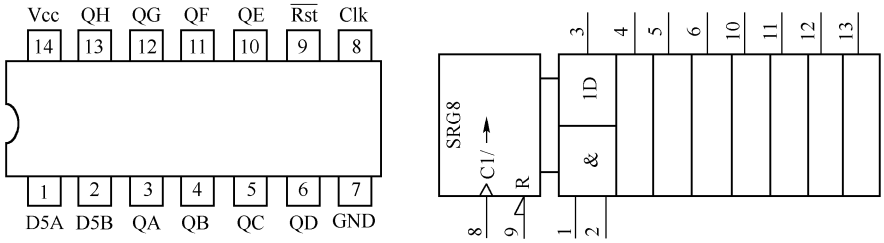


图 12-5　74LS164 外形及逻辑符号图

〖**74LS164 的基本知识**〗 74LS164 属于高速硅门 CMOS 器件，与低功耗肖特基型 TTL 器件的引脚兼容。74LS164 是 8 位边沿触发式移位寄存器，串行输入数据，然后并行输出，它有 14 个引脚，其外形及逻辑符号如图 11-5 所示，各引脚功能如表 12-4。

表 12-4　74LS164 引脚功能

引　脚	名　称	说　明
1	DSA	数据输入
2	DSB	数据输入
3～6	QA～QD	输出
7	GND	地（0V）
8	Clk	时钟输入（低电平到高电平边沿触发）
9	$\overline{\text{Rst}}$	复位输入（低电平有效）
10～13	QE～QH	输出
14	Vcc	正电源

数据通过两个输入端（DSA 或 DSB）之一串行输入；任意输入端可以用做高电平使能端，控制另一输入端的数据输入。当 DSA、DSB 任意一个为低电平时，则禁止新数据输入。在使用时，通常将两个输入端或者连接在一起，或者把不用的输入端接高电平，一定不要悬空。

时钟（Clk）每次由低变高时，数据右移一位，输入到 QA，QA 是两个数据输入端（DSA 和 DSB）的逻辑与，它将上升时钟沿之前保持一个建立时间的长度。

主复位（$\overline{\text{Rst}}$）输入端为低电平时，将使其他所有输入端都无效，同时非同步地清除寄存器，强制所有的输出为低电平。



硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 12-5 所列的元器件，并完成图 12-6 所示的硬件电路图设计。

表 12-5　74LS164 驱动数码管显示可调数字钟所用元器件

单片机 ATmega16	电解电容 CAP-ELEC 10uF	瓷片电容 CAP22pF	电阻 RES
晶振 CRYSTAL 8MHz	8 位共阳数码管 7SEG-MPX8-CA-BLUE	串行移位 74LS164.IEC	按钮 BUTTON



软件设计

在项目十的任务 4 中使用字符式 LCD 作为时间显示，在此使用 74LS164 作为 8 位共阳数码管驱动芯片。因此，这两个任务中的主程序、T/C0、外部中断 0～外部中断 2 函数的工作流程基本相同，只不过在本设计中采用 SPI 总线方式，通过 74LS164 驱动 8 位 LED 数码管进行时间显示时，需考虑的问题主要是 SPI 数据的传输。

为了将时、分、秒的数据通过 SPI 传送到 74LS164 以进行时间显示时，在进行 SPI 数据传输前，需先将单片机设置为主机模式，然后编写一个数据 SPI 串行数据输出函数（SPI_MasterTransmit）和 LED 显示驱动函数（display）。在主程序中通过调用 display 函数即可。在 SPI_MasterTransmit 函数中，首先将需要输出的数据送到 SPDR，然后判断 SPIF 是否变为低电平（即等待传输结束），就可实现数据的输出。在 display 函数中，主要是通过多次设置 LED 片选信号，然后调用 SPI_MasterTransmit 函数将 hour_h、hour_l、minute_

h、minute_l、second_h、second_l 输出到 74LS164。SPI 数据传送流程图如图 12-7 所示。

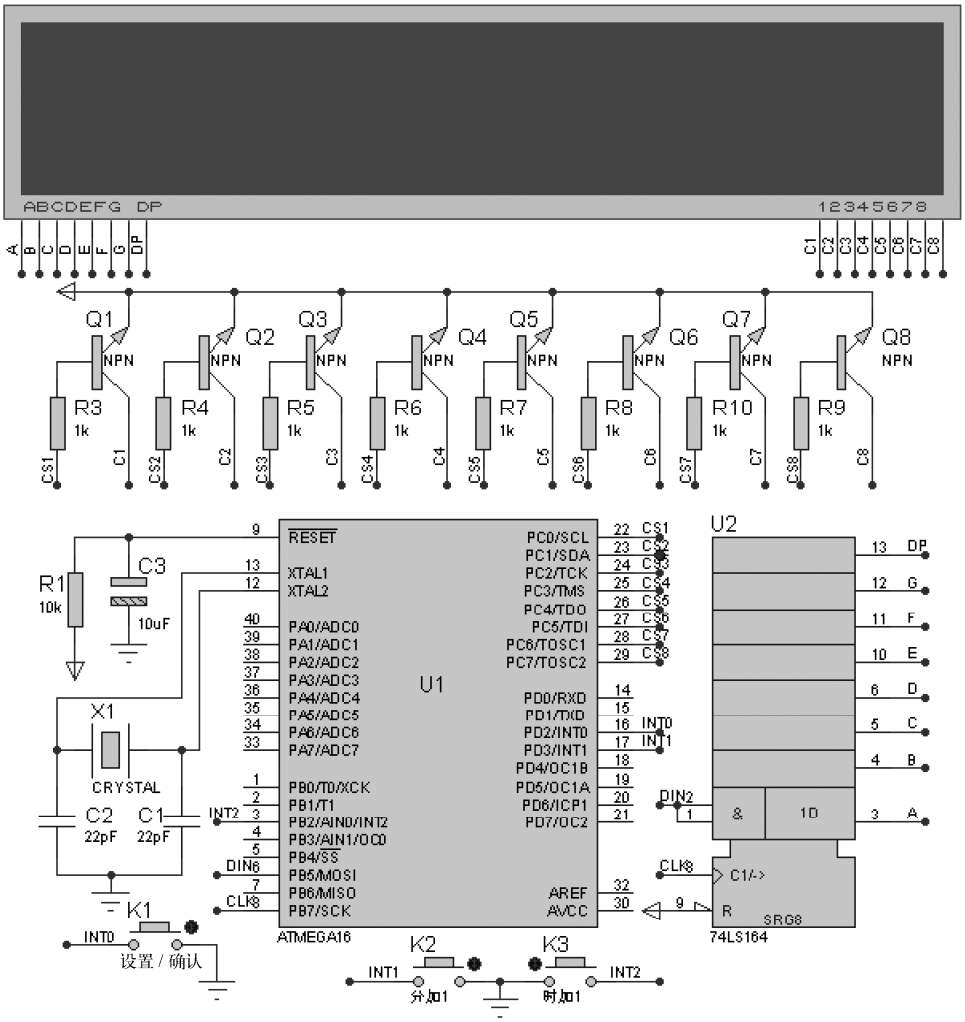


图 12-6　74LS164 驱动数码管显示可调数字钟电路图

源程序

```
/*****************************************************
  文件名:          74LS164 驱动数码管显示可调数字钟.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define   CS1        PORTC.0
#define   CS2        PORTC.1
```

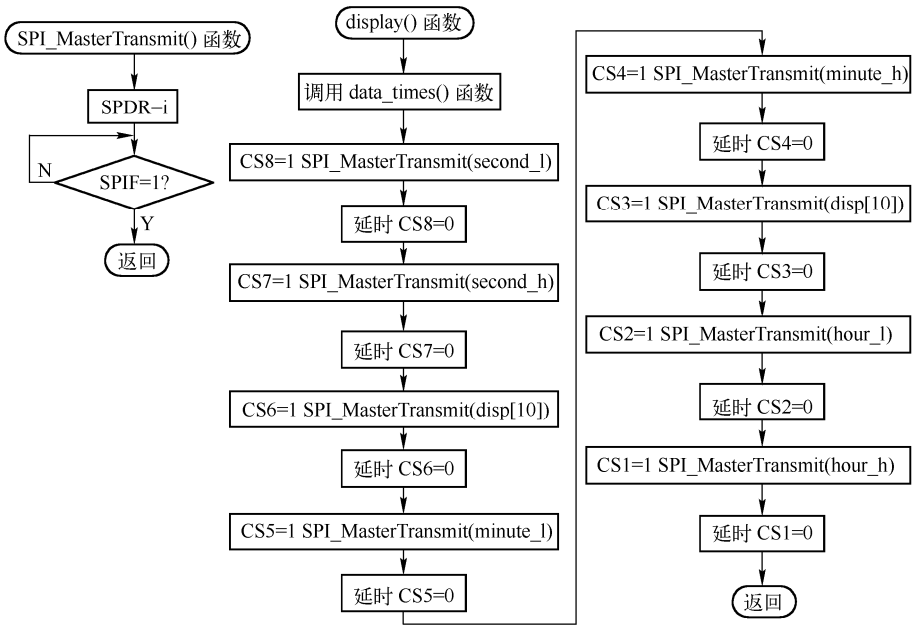


图 12-7　74LS164 驱动数码管显示可调数字钟数据传送流程图

```
#define   CS3        PORTC.2
#define   CS4        PORTC.3
#define   CS5        PORTC.4
#define   CS6        PORTC.5
#define   CS7        PORTC.6
#define   CS8        PORTC.7
#define   button1    PIND.2                    //K1 设置/确认键
#define   button2    PIND.3                    //K2 分加 1
#define   button3    PINB.2                    //K3 时加 1
uint   a,cnt;
uchar second,second_l,second_h;
uchar minute=18,minute_l,minute_h;
uchar hour=8,hour_l,hour_h;
uchar disp[] = {0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,0xBF};//0～9、-
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
interrupt[EXT_INT0] void int0_isr(void)         //INT0 中断函数
{
```

```
    delay(10);
    if(button1==0)                                  //再次确认按键，软件防抖
      {
        cnt++;
        if(cnt==1)
          {
            GICR=0xE0;                              //允许 INT0、INT1、INT2 产生中断
            TIMSK=0x00;                             //T/C2 停止计时
          }
        if(cnt>=2)
          {
            GICR=0x40;                              //允许 INT0 中断，INT1、INT2 中断禁止
            TIMSK=0x01;                             //T/C0 启动计时
            cnt=0;                                  //K1 按下值复位
          }
      }
}
interrupt[EXT_INT1] void int1_isr(void)             //INT1 中断函数
{
    delay(10);
    if(button2==0)                                  //再次确认按键，软件防抖
      {
        minute++;                                   //分加 1
      }
}
interrupt[EXT_INT2] void int2_isr(void)             //INT2 中断函数
{
    delay(10);
    if(button3==0)                                  //再次确认按键，软件防抖
      {
        hour++;                                     //时加 1
      }
}
interrupt [TIM0_OVF] void timer0_ovf_isr(void)      //T/C0 产生秒
{
    a++;
    TCNT0=0x00;
}
void    sec_in(void)                                //秒调整
{
    if(a==200)
      {
        second++;
        a=0;
```

```
        if(second==60)
          {
            second=0;
            minute++;
          }
      }
}
void min_in(void)                                    //分调整
{
  if(minute==60)
      {
        hour++;
        minute=0;
        second=0;
      }
}
void hour_in(void)                                   //时调整
{
   if(hour==24)
      {
        hour=0;
        minute=0;
        second=0;
      }
}
void disp_times(void)                                //时间显示整定
{
  second_l=second%10;
  second_h=second/10;
  minute_l=minute%10;
  minute_h=minute/10;
  hour_l=hour%10;
  hour_h=hour/10;
}
void SPI_MasterTransmit(unsigned char i)             //SPI 数据发送
{
  SPDR = i;                                          //启动数据串行输出
  while(!(SPSR==0x80))                               //SPIF 等待传输结束
      {
        ;
      }
}
void display(void)
{
```

```
        disp_times();
         CS8=1;
        SPI_MasterTransmit(disp[second_l]);
           delay(2);
           CS8=0;
           CS7=1;
        SPI_MasterTransmit(disp[second_h]);
           delay(2);
           CS7=0;
           CS6=1;
        SPI_MasterTransmit(disp[10]);
           delay(2);
           CS6=0;
           CS5=1;
        SPI_MasterTransmit(disp[minute_l]);
           delay(2);
           CS5=0;
           CS4=1;
        SPI_MasterTransmit(disp[minute_h]);
           delay(2);
           CS4=0;
           CS3=1;
        SPI_MasterTransmit(disp[10]);
           delay(2);
           CS3=0;
           CS2=1;
        SPI_MasterTransmit(disp[hour_l]);
           delay(2);
           CS2=0;
           CS1=1;
        SPI_MasterTransmit(disp[hour_h]);
           delay(2);
           CS1=0;
}
void INT_init(void)                                    //中断初始化设置
{
    MCUCR=0x0A;                                        //定义 INT0 和 INT1 为下降沿时产生中断
    MCUCSR=0x00;                                       //定义 INT2 为下降沿时产生中断
     GICR=0x40;                                        //只允许 INT0 产生中断
     #asm("sei")                                       //开启总中断
}
void TC0_init(void)                                    //T/C0 初始化设置
{
   TCCR0=0x05;              //WGM01=0、WGM00=0，选择普通模式；COM01=0、COM00=0，
```

```
                                //使 PB3 为通用 I/O（OC0 不与引脚相连）；CS02=1、CS01=0、
                                //CS01=1，选择 1024 分频
    TCNT0=0x00;
     OCR0=0x00;
    TIMSK=0x01;                 //T/C0 启动计时
}
void port_init(void)            //端口初始化
{
      DDRB=0xFB;                //PB2 输入，PB5、PB7 输出
     PORTC=0xFF;
      DDRC=0xFF;
     PORTC=0xFF;
      DDRD=0xF3;                //PD2、PD3 输入
     PORTD=0xFF;
}
void SPI_MasterInit(void)       //SPI 总线发送初始化
 {
    SPCR=0x53;                  //SPE=1，MSTR=1，SPR1=1，SPR0=1，使能 SPI 主机模
                                //式，设置时钟速率为 fck/128
 }
void main(void)
{
   port_init();
   INT_init();
   TC0_init();
   SPI_MasterInit();
   while(1)
      {
        sec_in();
        min_in();
        hour_in();
        display();
      }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“74LS164 驱动数码管显示可调数字钟.COFF”文件，然后在 Proteus ISIS 中打开已创建的 74LS164 驱动数码管显示可调数字钟电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试运行状态下，LED 数码管显示的时间初值为 08–18–00，而后在此基础上进行计时。第 1 次按下按键 K1 后，可按下按键 K2 或 K3 进行分、时调整，第 2 次按下按键 K1 后，在调整好时间的基础上进行计时，其仿真效果如图 12–8 所示。

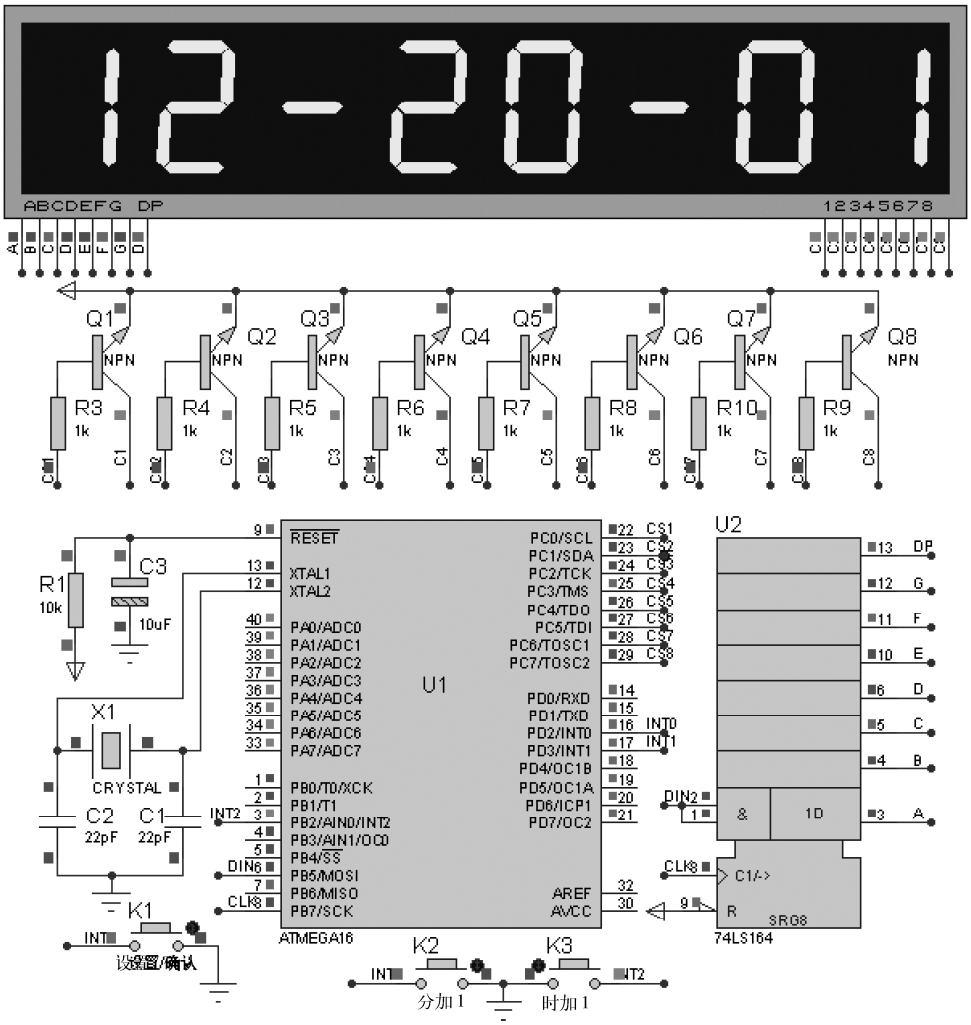


图 12-8　74LS164 驱动数码管显示可调数字钟仿真效果图

任务 3　74HC595 驱动 1 位数码管显示

使用 74HC595 控制 1 位共阳极数码管循环显示 0～9，要求 74HC595 由 ATmega16 的 SPI 总线接口控制。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 12-6 所列的元器件，并完成图 12-9 所示的硬件电路图设计。

表 12-6　74HC595 驱动 1 位数码管显示所用元器件

单片机 ATmega16	电解电容 CAP-ELEC 10uF	瓷片电容 CAP22pF	电阻 RES
晶振 CRYSTAL 8MHz	共阳数码管 7SEG-MPX-CA-GRN	串行移位 74HC595	

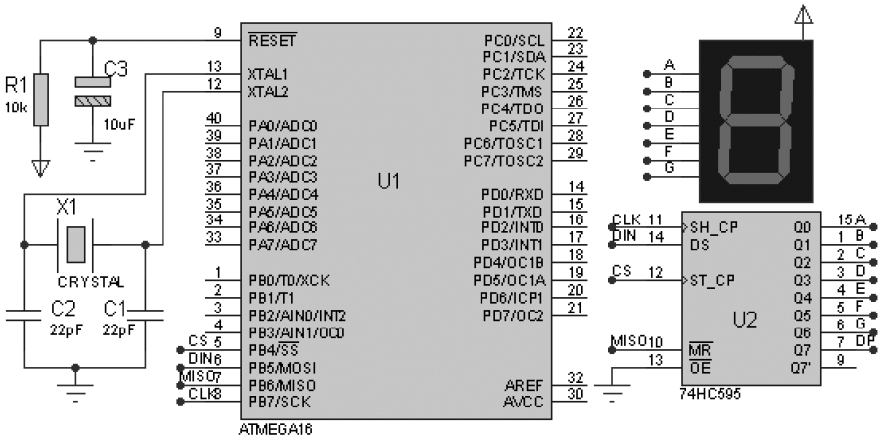


图 12-9　74HC595 驱动 1 位数码管显示电路图

软件设计

在实际系统中，LED 数码管的显示使用 74HC595 驱动的较多，对于没有 SPI 接口的单片机而言，通常采用模拟 SPI 串口的方式进行，即根据时序来传送数据，在此不细述。本任务直接使用 ATmega16 的 SPI 总线接口来驱动 LED 数码管时，应编写 SPI 串行数据输出函数（SPI_MasterTransmit），和 LED 显示驱动函数（display）。在主程序中通过调用 display 函数即可。其程序流程图如图 12-10 所示。

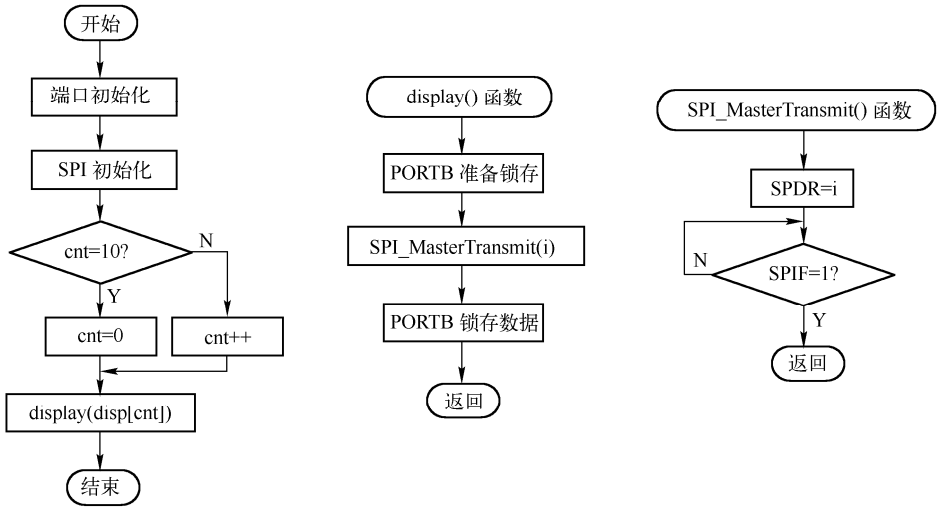


图 12-10　74HC595 驱动 1 位数码管显示程序流程图

源程序

```
/*****************************************************
  文件名:          74HC595 驱动 1 位数码管显示.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
```

```
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
uchar disp[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,0xBF};//0～9、
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
void SPI_MasterTransmit(uchar i)                    //SPI 数据发送
{
  SPDR = i;                                         //启动数据串行输出
  while(!(SPSR==0x80))                              //SPIF 等待传输结束
    {
      ;
    }
}
void port_init(void)                                //端口初始化
{
     DDRB=0xFB;                                     //PB2 输入，PB5、PB7 输出
   PORTB=0xF0;
}
void SPI_MasterInit(void)                           //SPI 总线发送初始化
 {
   SPCR=0x53;                                       //SPE=1，MSTR=1，SPR1=1，SPR0=1，使能
                                                    //SPI 主机模式，设置时钟速率为 fck/128
 }
void display(uchar i)
{
  PORTB &= ～(1 << 4);                              //准备锁存
  SPI_MasterTransmit(i);
  PORTB |= (1 << 4);                                //锁存数据
}
void main(void)
{
  uint   cnt;
  port_init();
  SPI_MasterInit();
  while(1)
    {
     for(cnt=0;cnt<10;cnt++)
         {
             display(disp[cnt]);
             delay(500);
```

```
        }
    }
}
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“74HC595 驱动 1 位数码管显示.COFF”文件，然后在 Proteus ISIS 中打开已创建的 74HC595 驱动 1 位数码管显示电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试运行状态下，LED 数码管每隔 0.5s 循环显示数字 0～9，其仿真效果如图 12-11 所示。

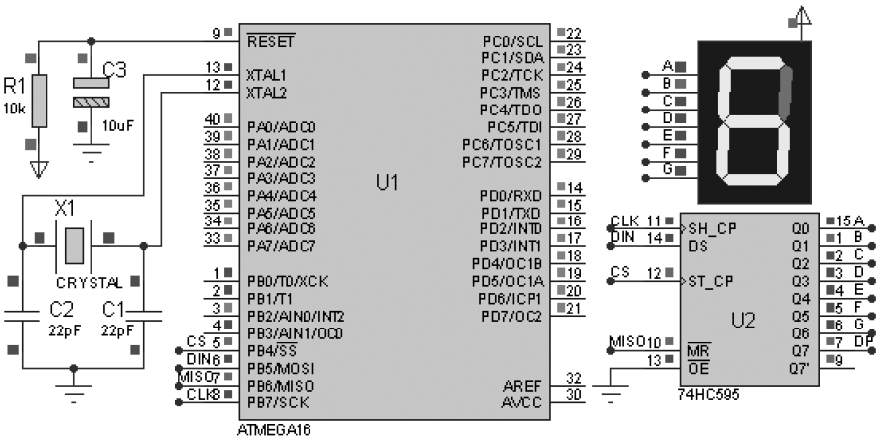


图 12-11　74HC595 驱动 1 位数码管显示仿真效果图

任务 4　74HC595 驱动数码管动态显示

在项目十一的任务 4 中，采用了 PORT 端口和 PORTB 端口驱动 4 位数码管显示电压值，现改用 74HC595 驱动 4 位共阳极数码管动态显示电压值，要求 74HC595 由 ATmega16 的 SPI 总线接口控制。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 12-7 所列的元器件，并完成图 12-12 所示的硬件电路图设计。

表 12-7　74HC595 驱动数码管动态显示所用元器件

单片机 ATmega16	电解电容 CAP-ELEC 10uF	瓷片电容 CAP22pF	电阻 RES
晶振 CRYSTAL 8MHz	共阳数码管 7SEG-MPX8-CA-BLUE	串行移位 74HC595	

软件设计

74HC595 驱动数码管动态显示电压值，其电压转换部分在项目十一的任务 4 中已经讲

述，在此不赘述。使用 74HC595 驱动 4 位共阳极数码管动态显示时，采用动态扫描法，根据时序分别将段码送给相应的 LED 数码管，并且该位 LED 数码管的 CS 有效即可，其动态显示流程图如图 12-13 所示。

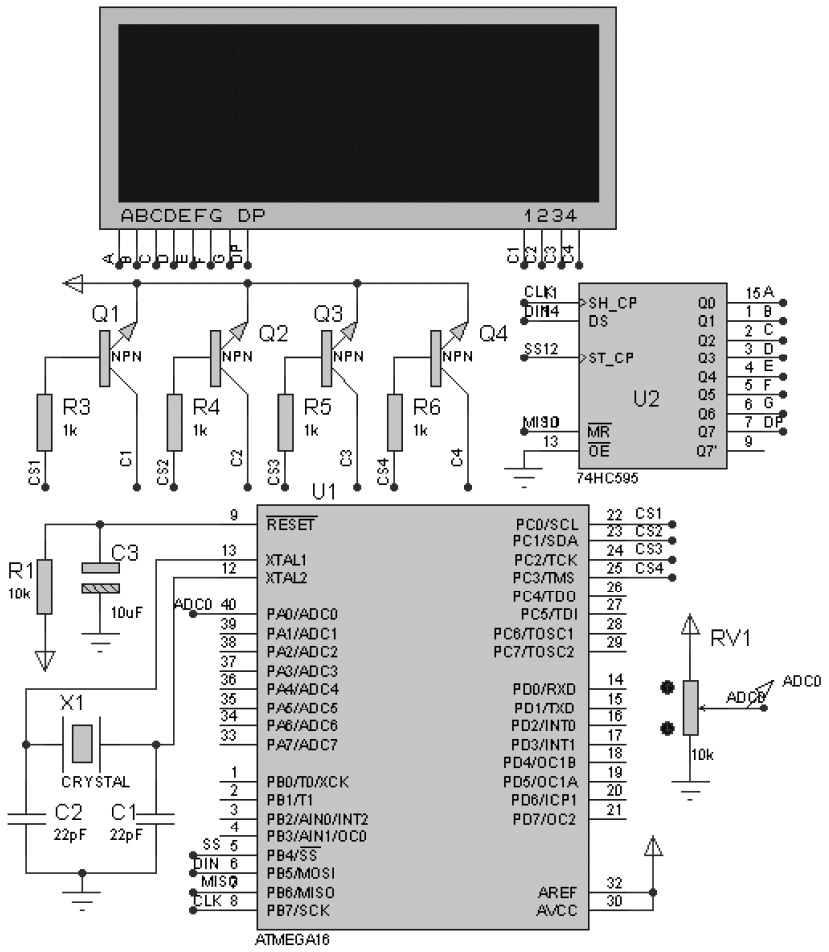


图 12-12　74HC595 驱动数码管动态显示电路图

源程序

```
/*****************************************************
  文件名:          74HC595 驱动数码管动态显示.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define ulong unsigned long
#define   CS1   PORTC.0
#define   CS2   PORTC.1
#define   CS3   PORTC.2
```

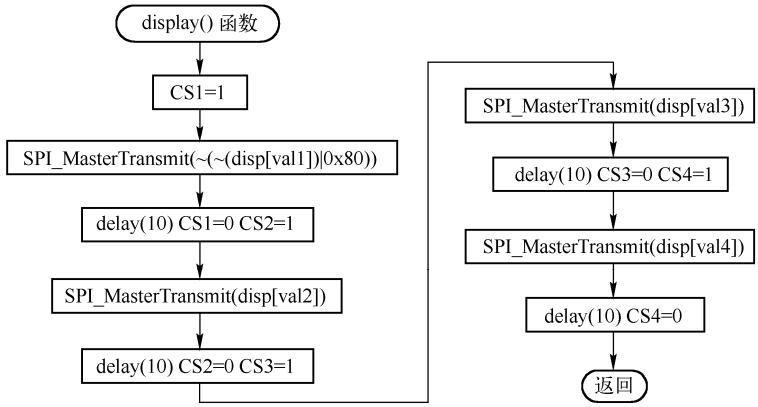


图 12-13　74HC595 驱动数码管动态显示程序流程图

```
#define   CS4   PORTC.3
uchar disp[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,0xBF};          //0～9、
uint adc_v,val1,val2,val3,val4;
uint cnt;
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
interrupt [TIM0_COMP] void timer0_comp_isr(void)      //T/C0 比较匹配中断服务
{
   cnt=0x01;
}
interrupt [ADC_INT] void adc_isr(void)                //ADC 转换完成中断服务
{
     uint adc_data;
     adc_data=ADCW;                                   //读取 ADC 转换结果
     adc_v=(ulong)adc_data*5000/1024;                 //换算成电压值
}
void volt_in(void)                                    //电压值各位分离
{
  val1=adc_v/1000;
  val2=(adc_v/100)%10;
  val3=(adc_v/10)%10;
  val4=adc_v%10;
}
void SPI_MasterTransmit(uchar i)                      //SPI 数据发送
{
```

```
    SPDR = i;                    //启动数据串行输出
    while(!(SPSR==0x80))         //SPIF 等待传输结束
       {
         ;
       }
}
void port_init(void)             //端口初始化
{
      DDRA=0x00;                 //模拟量由 PA0 输入
     PORTA=0x00;
      DDRB=0xFF;                 //PB2 输入，PB5、PB7 输出
     PORTB=0xF0;
      DDRC=0xFF;
     PORTC=0x00;
}
void SPI_MasterInit(void)        //SPI 总线发送初始化
 {
    SPCR=0x53;                   //SPE=1，MSTR=1，SPR1=1，SPR0=1，使能 SPI 主机模式，
                                 //设置时钟速率为 fck/128
 }
void disp_595(uchar i)
{
   PORTB &= ～(1 << 4);          /*准备锁存*/
   SPI_MasterTransmit(i);
   PORTB |= (1 << 4);            /*锁存数据*/
}
void ADC_init(void)
 {
    ADMUX=0x40;                  //参考电源 AVcc、ADC0 单端输入
    SFIOR&=0x1F;
    SFIOR|=0x60;                 //选择 T/C0 比较匹配中断为 ADC 触发源
    ADCSRA=0xAE;                 //ADC 允许、自动触发转换、ADC 转换中断允许、ADCclk =
                                 //8000000Hz/64=125kHz
    #asm("sei")                  //开放全局中断
 }
   void timer0_init(void)
 {
    TCCR0=0x0B;                  //内部时钟，64 分频（8M/64=125kHz），CTC 模式
    TCNT0=0x00;
    OCR0=0xF9;                   //OCR0 = 0xF9(即 249)，(249+1)/125=2ms
    TIMSK=0x02;                  //允许 T/C0 比较中断
 }
void display(void)
{
   uchar a;
```

```
    a=～disp[val1];
    CS1=1;
    disp_595(～(a|0x80));
    delay(10);
    CS1=0;
    CS2=1;
    disp_595(disp[val2]);
    delay(10);
    CS2=0;
    CS3=1;
    disp_595(disp[val3]);
    delay(10);
    CS3=0;
    CS4=1;
    disp_595(disp[val3]);
    delay(10);
    CS4=0;
}
void main(void)
{
    port_init();
    timer0_init();
    SPI_MasterInit();
    ADC_init();
    while(1)
      {
        if(cnt==1)
          {
            volt_in();
            display();
            cnt=0;
          }
      }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“74HC595 驱动数码管动态显示.COFF”文件，然后在 Proteus ISIS 中打开已创建的 74HC595 驱动数码管动态显示电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。滑动可调电阻，可以观看 4 位共阳极数码管显示的电压值也会有所变化，如图 12-14 所示。多次滑动可调电阻，可以看出，LED 数码管显示的电压值与电压探针显示的电压值略有不同，只不过两个值的误差较小，用户可以认为 LED 数码管显示的电压值即为实际所测电压值。

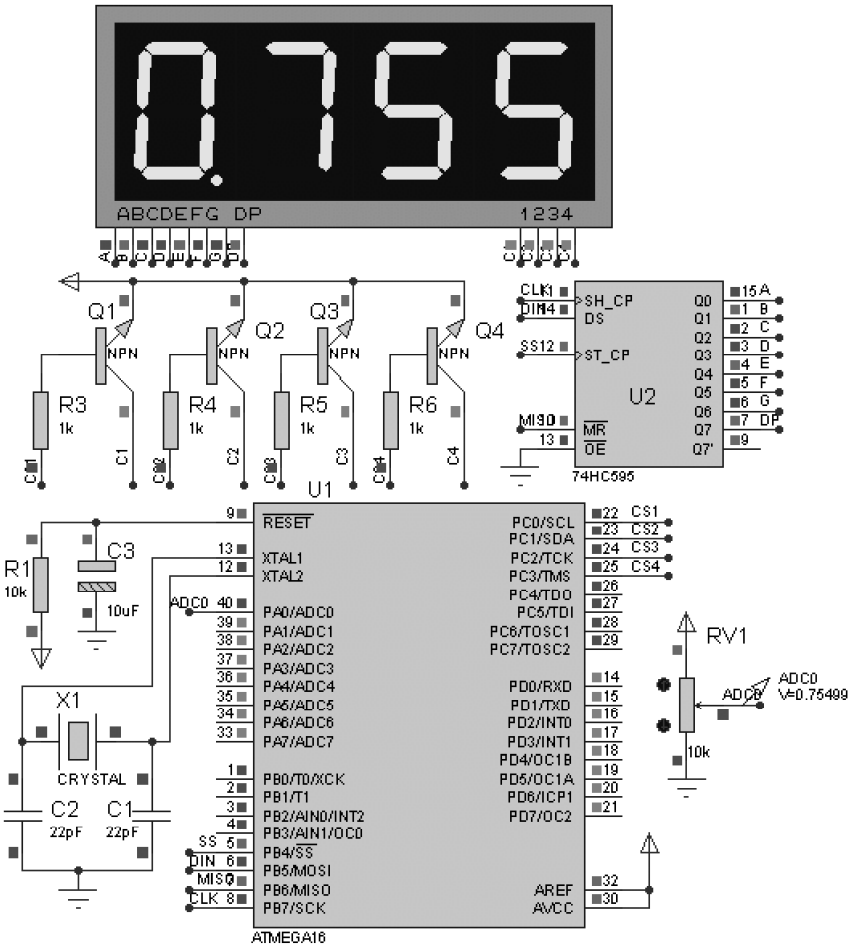


图 12-14　74HC595 驱动数码管动态显示仿真效果图

任务 5　74HC595 驱动 8 位数码管静态显示

使用 74HC595 驱动 8 位 LED 共阳极数码管进行静态显示，显示初值为“12518623”，每按一次按键 K 时，8 位 LED 数码管显示内容循环左移 1 位。要求 74HC595 由 ATmega16 的 SPI 总线接口控制。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 12-8 所列的元器件，并完成图 12-15 所示的硬件电路图设计。在图中，只绘制出了 74HC595 芯片及 LED 数码管部分，单片机最小部分未绘制出。U2(74HC595)的 SH_CP 与 PB7 连接；DIN 与 PB5 连接；RCK 与 PB4 连接；$\overline{\text{MR}}$ 与 PB6 连接；PD2 与按键 K 的一端连接，而按键的另一端接地。

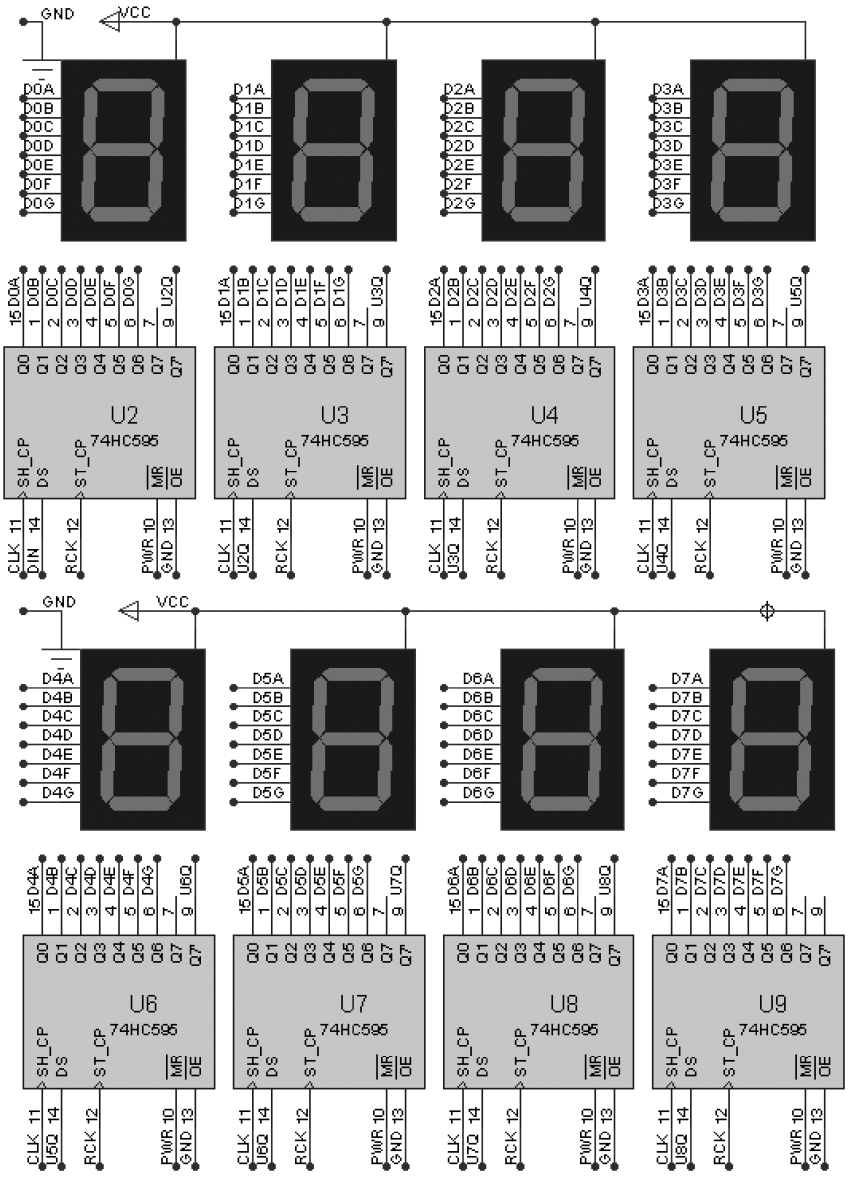


图 12-15　74HC595 驱动数码管静态显示电路图

表 12-8　74HC595 驱动数码管静态显示所用元器件

单片机 ATmega16	电解电容 CAP-ELEC 10uF	瓷片电容 CAP22pF	电阻 RES
晶振 CRYSTAL 8MHz	共阳数码管 7SEG-MPX-CA-BLUE	串行移位 74HC595	按钮 BUTTON

软件设计

进行静态显示时，ATmega16 单片机将每位显示数据通过 SPI 串行总线由 74HC595 串行移位输出即可，因此在程序中需编写 SPI 串行数据输出函数（SPI_MasterTransmit）、LED 数码管移位控制函数（LED_move()）、INT0 中断服务函数等，其程序流程图如图 12-16 所示。

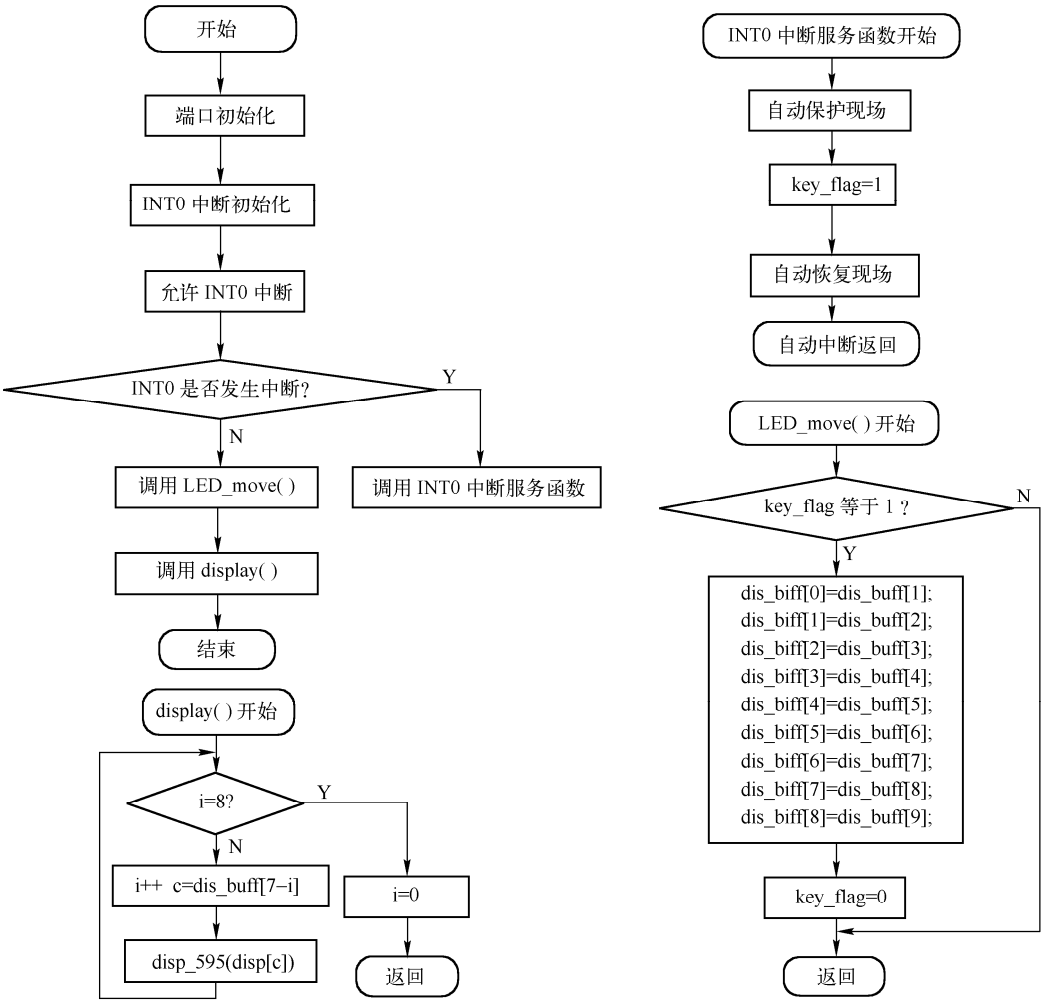


图 12-16　74HC595 驱动数码管静态显示程序流程图

源程序

```
/*****************************************************
  文件名:          74HC595 驱动数码管静态显示.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define ulong unsigned long
#define   CS1             PORTC.0
#define   CS2             PORTC.1
#define   CS3             PORTC.2
#define   CS4             PORTC.3
#define   button          PIND.2
```

```
uchar disp[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,0xBF};//0～9、
uchar dis_buff[9]={1,2,5,1,8,6,2,3,1};                    //定义显示初值
bit   key_flag;                                           //是否按下按键的状态标志位
void delay(uint k)
{
  uint m,n;
  for(m=0;m<k;m++)
    {
      for(n=0;n<1140;n++)
        {;}
    }
}
interrupt[EXT_INT0] void int0_isr(void)                   //INT0 中断函数
{
  delay(10);
  if(button==0)                                           //再次确认按键，软件防抖
    {
      key_flag=1;
    }
}
void SPI_MasterTransmit(uchar i)                          //SPI 数据发送
{
  SPDR = i;                                               //启动数据串行输出
  while(!(SPSR==0x80))                                    //SPIF 等待传输结束
    {
      ;
    }
}
void port_init(void)                                      //端口初始化
{
     DDRA=0x00;                                           //模拟量由 PA0 输入
    PORTA=0x00;
     DDRB=0xFF;
    PORTB=0xF0;
     DDRC=0xFF;
    PORTC=0x00;
     DDRD=0xFB;
    PORTD=0xFF;
}
void INT0_init(void)                                      //中断初始化设置
{
    MCUCR=0x02;                                           //INT0 为下降沿时产生中断请求
     GICR=0x60;                                           //允许 INT0 产生中断
     #asm("sei")                                          //使能全局中断
}
void SPI_MasterInit(void)                                 //SPI 总线发送初始化
 {
    SPCR=0x53;                                            //SPE=1，MSTR=1，SPR1=1，SPR0=1，使能
```

```
                                                    //SPI 主机模式，设置时钟速率为 fck/128
 }
void disp_595(uchar i)
{
  PORTB &= ～(1 << 4);                              /*准备锁存*/
  SPI_MasterTransmit(i);
  PORTB |= (1 << 4);                                /*锁存数据*/
}
void   LED_move (void)
{
  if(key_flag==1)                                   //有按键按下时，LED 数码管显示左移 1 位
     {
        dis_buff[0]=dis_buff[1];
        dis_buff[1]=dis_buff[2];
        dis_buff[2]=dis_buff[3];
        dis_buff[3]=dis_buff[4];
        dis_buff[4]=dis_buff[5];
        dis_buff[5]=dis_buff[6];
        dis_buff[6]=dis_buff[7];
        dis_buff[7]=dis_buff[8];
        dis_buff[8]=dis_buff[0];
        key_flag=0;
     }
}
void display(void)
{
  uchar   i,c;
  for(i=0;i<8;i++)
    {
      c=dis_buff[7-i];
      disp_595(disp[c]);
    }
}
void main(void)
{
  port_init();
  INT0_init();
  SPI_MasterInit();
  while(1)
    {
      LED_move();
      display();
      delay(100);
    }
}
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“74HC595 驱动数码管静态显示.COFF”文件，然后在 Proteus ISIS 中打开已创建的 74HC595 驱动数码管静态显示电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。滑动可调电阻，可以观看 4 位共阳极数码管显示的电压值也会有所变化，如图 12-17 所示。在按下按键 K 前，8 位 LED 数码管显示的初值为“12518623”。每按一次按键 K，8 位 LED 数码管显示的内容循环左移 1 位，其仿真效果如图 12-17 所示。

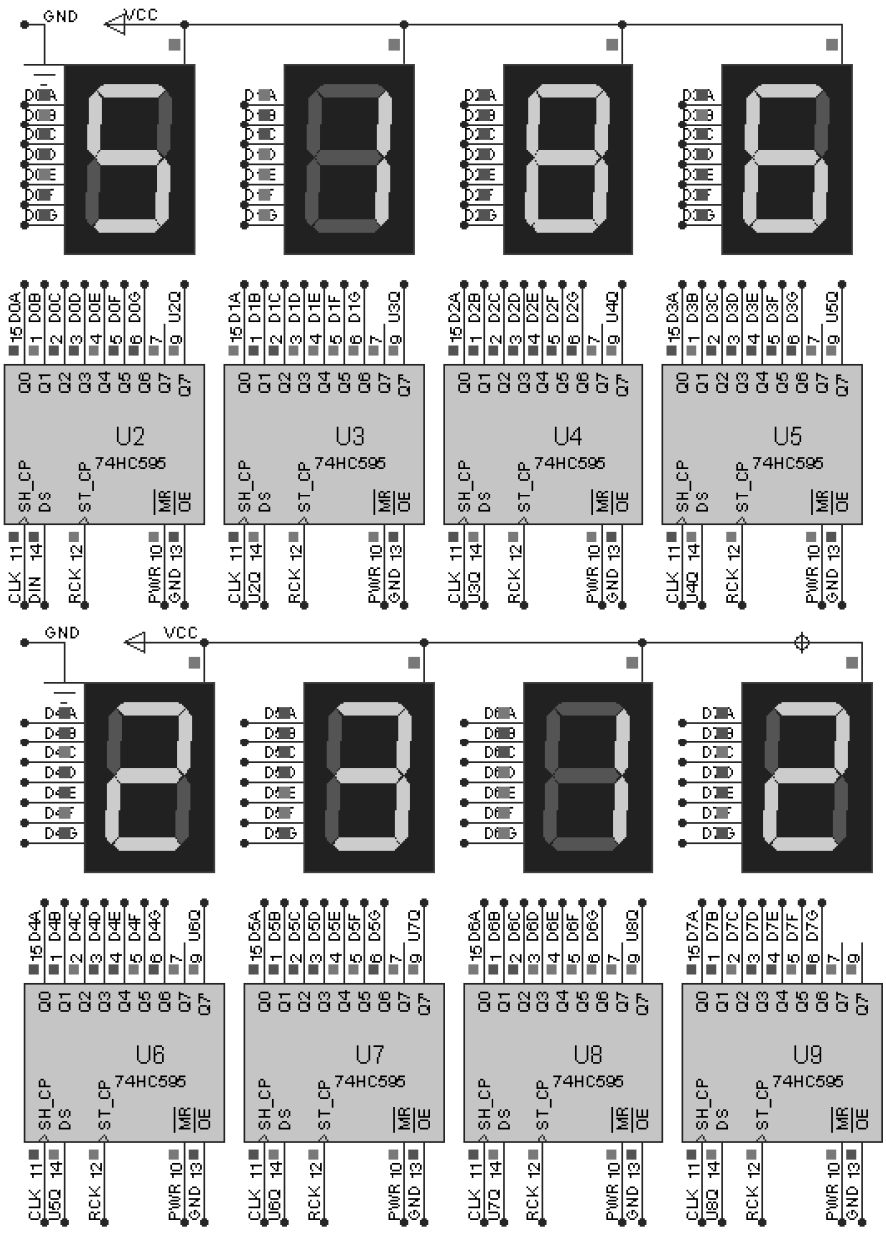


图 12-17　74HC595 驱动数码管静态显示仿真效果图

任务 6　MAX7219 驱动数码管动态显示

使用 MAX7219 串行驱动 8 位共阴极 LED 数码管，动态显示为 12518623。

〖**MAX7219 基础知识**〗在单片机应用系统中，通常需要 LED 显示器对系统的状态进行观察。一般情况下，LED 显示器的显示方式有静态显示和动态显示两种。无论是静态显示还是动态显示，单片机都工作在并行 I/O 口状态或存储器方式中，需要占用比较多的 I/O 口线。如果采用 MAX7219 作为 LED 显示器的接口电路，则只需占用单片机的 3 根线就可串行实现 8 位 LED 的显示驱动和控制。

MAX7219 是美国 MAXIM（美信）公司生产的串行 I/O 共阴极显示驱动器。采用 3 线制（SPI）串行接口技术进行数据的传送，可直接与单片机连接，用户能方便地修改内部参数实现多位 LED 显示器的显示。MAX7219 片内含有硬件动态扫描显示控制，每块芯片可驱动 8 个 LED 显示器。

【MAX7219 外部封装及引脚功能】MAX7219 是 7 段共阴极 LED 显示器的驱动器，采用 24 引脚的 DIP 和 SO 两种封装形式，其外形封装如图 12-18 所示。

MAX7219 LED 驱动器各引脚功能如下所述。

☺ Din：串行数据输入端。在 CLK 的上升沿，数据被锁入 16 位内部移位寄存器中。

☺ DIG0~DIG7：8 位数码管驱动线，输出位选信号，从数码管的共阴极吸收电流。

☺ GND：地线。

☺ LOAD：装载数据控制端。在 LOAD 的上升沿，最后送入的 16 位串行数据被锁存到移位寄存器中。

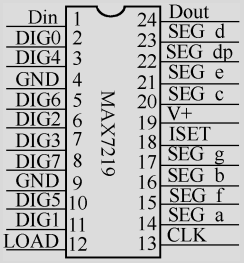


图 12-18　MAX7219 外形封装

☺ CLK：串行时钟输入端。最高输入频率为 10MHz，在 CLK 的上升沿，数据被送入内部移位寄存器；在 CLK 的下降沿，数据 Dout 端输出。

☺ SEG a~SEG g：LED 七段显示器段驱动端。用于驱动当前 LED 段码。

☺ SEG dp：小数点驱动端。

☺ ISET：LED 段峰值电流设置端。ISET 端通过一只电阻与电源 V+相连，调节电阻值，改变 LED 段提供峰值电流。

☺ V+：+5V 电源。

☺ Dout：串行数据输出端。进入 Din 的数据在 16.5 个时钟后送到 Dout 端，Dout 在级联时传送到下一片 MAX7219 的 Din 端。

【MAX7219 内部结构】MAX7219 的内部结构如图 12-19 所示。主要由段驱动器、段电流基准、二进制 ROM、数位驱动器、5 个控制寄存器、16 位移位寄存器、8×8 双端口 SRAM、地址寄存器和译码器、亮度脉宽调制器、多路扫描电路等部分组成。

数位驱动器用于选择某位 LED 显示。串行数据以 16 位数据包的形式从 Din 引脚输入，在 CLK 的每个上升沿时，不管 LOAD 引脚的工作状态如何，数据逐位地串行送入片内 16 位移位寄存器中。在第 16 个 CLK 上升沿出现的同时或之后，在下一个 CLK 上升沿前，LOAD 必须变为高电平，否则移入移位寄存器的数据将会被丢失。这

16 位数据包见表 12-9。从表中可以看出 D_{15}~D_{12} 为无关位，取任意值，通常全为“1”，D_{11}~D_8 为 4 位地址，D_7~D_0 为 5 个控制寄存的命令字或 8 位 LED 待显示的数据位，在 8 位数据中 D_7 为最高位，D_0 为最低位。一般情况下，程序先送控制命令，再送数据到显示寄存器，但必须每 16 位为一组，从最高位开始送数据，一直送到最低位为止。

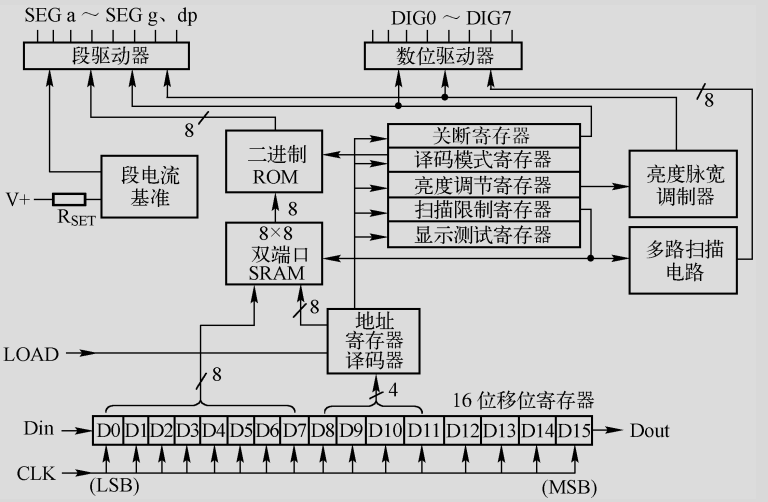


图 12-19　MAX7219 内部结构

表 12-9　16 位数据包格式

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
×	×	×	×	地　址				MSB			数	据		LSB	

通过对 D11~D8 中 4 位地址译码，可寻址 14 个内部寄存器，即 8 个数位寄存器、5 个控制寄存器及 1 个空操作寄存器。14 个内部寄存器地址见表 12-10。空操作寄存器主要用于多个 MAX7219 级联，允许数据通过而不对当前 MAX7219 产生影响。

表 12-10　14 个内部寄存器地址

寄存器	地　址					十六进制代码
	D15～D12	D11	D10	D9	D8	
空操作	×	0	0	0	0	X0
DIG0	×	0	0	0	1	X1
DIG1	×	0	0	1	0	X2
DIG2	×	0	0	1	1	X3
DIG3	×	0	1	0	0	X4
DIG4	×	0	1	0	1	X5
DIG5	×	0	1	1	0	X6
DIG6	×	0	1	1	1	X7
DIG7	×	1	0	0	0	X8
译码模式	×	1	0	0	1	X9
亮度调节	×	1	0	1	0	XA

续表

寄存器	地　址					十六进制代码
	D15～D12	D11	D10	D9	D8	
扫描限制	×	1	0	1	1	XB
关断模式	×	1	1	0	0	XC
显示测试	×	1	1	1	1	XF

5 个控制寄存器分别是译码模式寄存器、亮度调节寄存器、扫描限制寄存器、关断模式寄存器、显示测试寄存器。在使用 MAX7219 时，首先必须对 5 个控制寄存器进行初始化。5 个控制寄存器的设置含义如下所述。

☺ 译码模式选择寄存器（地址：0xX9）：决定数位驱动器的译码方式，共有 4 种译码模式选择。每一位对应一个数位。其中，“1”代表 B 码方式；“0”表示不译方式。驱动 LED 数码管时，应将数位驱动器设置为 B 码方式。一般情况下，应将数据位置为全“0”，即选择“全非译码方式”，在此方式下，8 个数据位分别对应 7 个段和小数点。

当选择译码模式时，译码器只对数据的低 4 位进行译码（D3~D0），D4~D6 为无效位。D7 位用来设置小数点，不受译码器的控制且为高电平。表 12-11 所列为 B 型译码的格式。

表 12-11　B 型译码格式

字符代码	寄存器数据						段　码							
	D7	D6～D4	D3	D2	D1	D0	DP	G	F	E	D	C	B	A
0		×	0	0	0	0		1	1	1	1	1	1	0
1		×	0	0	0	1		0	1	1	0	0	0	0
2		×	0	0	1	0		1	1	0	1	1	0	1
3		×	0	0	1	1		1	1	1	1	0	0	1
4		×	0	1	0	0		0	1	1	0	0	1	1
5		×	0	1	0	1		1	0	1	1	0	1	1
6		×	0	1	1	0		1	0	1	1	1	1	1
7		×	0	1	1	1		1	1	1	0	0	0	0
8		×	1	0	0	0		1	1	1	1	1	1	1
9		×	1	0	0	1		1	1	1	1	0	1	1
-		×	1	0	1	0		0	0	0	0	0	0	1
E		×	1	0	1	1		1	0	0	1	1	1	1
H		×	1	1	0	0		0	1	1	0	1	1	1
L		×	1	1	0	1		0	0	0	1	1	1	0
P		×	1	1	1	0		1	1	0	0	1	1	1
blank		×	1	1	1	1		0	0	0	0	0	0	0

当选择不译码时，数据的 8 位与 MAX7219 的各段线上的信号一致，表 12-12 列出了数字对应的段码位。

表 12-12　每个数字对应的段码位

	寄存器数据							
	D7	D6	D5	D4	D3	D2	D1	D0
	DP	g	f	e	d	c	b	a

☺ 亮度调节寄存器（地址：×A）：用于 LED 数码管显示亮度强弱的设置。利用其 D_3~D_0 位控制内部亮度脉宽调制器 DAC 的占空比来控制 LED 段电流的平均值，实现 LED 的亮度控制。D_3~D_0 取值范围为 0000~1111，对应电流的占空比则从 1/32、3/32 变化到 31/32，共 16 级，D_3~D_0 的值越大，LED 显示越亮。而亮度控制寄存器中的其他各位未使用，可置任意值。亮度调节寄存器的设置格式见表 12-13。

表 12-13　亮度调节寄存器中的设置格式

占空比	D7	D6	D5	D4	D3	D2	D1	D0	十六进制代码
1/32	x	x	x	x	0	0	0	0	0xX0
3/32	x	x	x	x	0	0	0	1	0xX1
5/32	x	x	x	x	0	0	1	0	0xX2
7/32	x	x	x	x	0	0	1	1	0xX3
9/32	x	x	x	x	0	1	0	0	0xX4
11/32	x	x	x	x	0	1	0	1	0xX5
13/32	x	x	x	x	0	1	1	0	0xX6
15/32	x	x	x	x	0	1	1	1	0xX7
17/32	x	x	x	x	1	0	0	0	0xX8
19/32	x	x	x	x	1	0	0	1	0xX9
21/32	x	x	x	x	1	0	1	0	0xXA
23/32	x	x	x	x	1	0	1	1	0xXB
25/32	x	x	x	x	1	1	0	0	0xXC
27/32	x	x	x	x	1	1	0	1	0xXD
29/32	x	x	x	x	1	1	1	0	0xXE
31/32	x	x	x	x	1	1	1	1	0xXF

☺ 扫描限制寄存器（地址：×B）：用于设置显示数码管的个数（1~8）。该寄存器的 D_2~D_0（低 3 位）指定要扫描的位数，D_7~D_3 无关，支持 0~7 位，各数位均以 1.3kHz 的扫描频率被分路驱动。当 D2~D0=111 时，可接 8 个数码管。扫描限制寄存器的设置格式见表 12-14。

表 12-14　扫描限制寄存器的设置格式

扫描 LED 位数	D7	D6	D5	D4	D3	D2	D1	D0	十六进制代码
只扫描 0 位	x	x	x	x	x	0	0	0	0xX0
扫描 0 或 1 位	x	x	x	x	x	0	0	1	0xX1
扫描 0，1，2 位	x	x	x	x	x	0	1	0	0xX2
扫描 0，1，2，3 位	x	x	x	x	x	0	1	1	0xX3
扫描 0，1，2，3，4 位	x	x	x	x	x	1	0	0	0xX4
扫描 0，1，2，3，4，5 位	x	x	x	x	x	1	0	1	0xX5
扫描 0，1，2，3，4，5，6 位	x	x	x	x	x	1	1	0	0xX6
扫描 0，1，2，3，4，5，6，7 位	x	x	x	x	x	1	1	1	0xX7

☺ 关断模式寄存器（地址：×C）：用于关断所有显示器。有 2 种选择模式：D_0= “0”，

关断所有显示器，但不会消除各寄存器中保持的数据；D_0= “1”，正常工作状态。剩下各位未使用，可取任意值。通常情况下选择正常操作状态。

☺ 显示测试寄存器（地址：×F）：用于检测外接 LED 数码管是工作在测试状态还是正常操作状态。D_0= “0”，LED 处于正常工作状态；D_0= “1”，LED 处于显示测试状态，所有 8 位 LED 各位全亮，电流占空比为 31/32。D_7~D_1 位未使用，可任意取值。一般情况下选择正常工作状态。

【工作时序】 MAX7219 工作时序如图 12-20 所示。从图中可以看出，在 CLK 的每个上升沿，都有一位数据从 Din 端输入，加载到 16 位移位寄存器中。在 LOAD 的上升沿，输入的 16 位串行数据被锁存到数位或控制寄存器中。LOAD 必须在第 16 个 CLK 上升沿出现的同时或在下一个 CLK 上升沿之前，变为高电平，否则移入移位寄存器的数据将会被丢失。

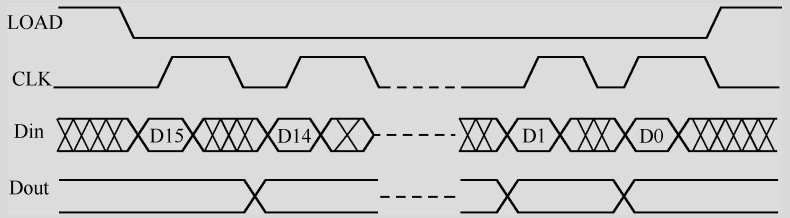


图 12-20　MAX7219 工作时序



硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 12-15 所列的元器件，并完成图 12-21 所示的硬件电路图设计。

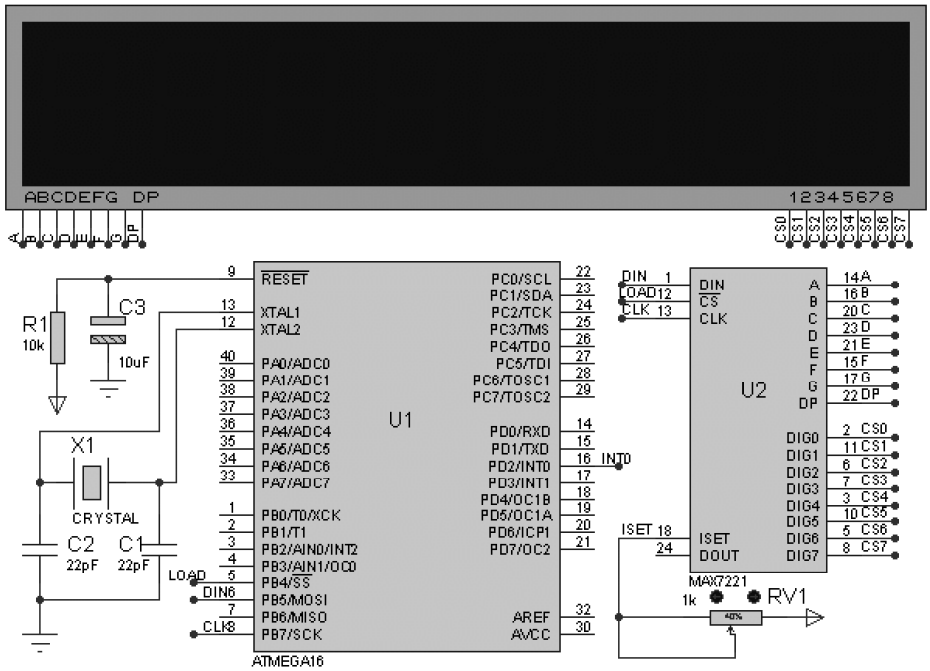


图 12-21　MAX7219 驱动数码管动态显示电路图

表 12-15　MAX7219 驱动数码管动态显示所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	晶振 CRYSTAL 8MHz	电解电容 CAP-ELEC
电阻 RES	可调电阻 POT-HG	数码管 7SEG-MPX8-CC-BLUE	LED 驱动 MAX7219

软件设计

无论是 MAX7219 的初始化，还是 8 个七段数码管的显示，均需对数据进行写入。16 位数据包分成两个 8 位的字节进行传送，第 1 字节是地址，第 2 字节是数据。在这 16 位数据包中，D_{15}～D_{12} 可以任意写，在此均置为“1”；D_{11}～D_8 决定所选通的内部寄存器地址；D_7～D_0 为待显示数据，8 个 LED 显示器的显示内容在 LED_code 中。其程序流程图如图 12-22 所示。

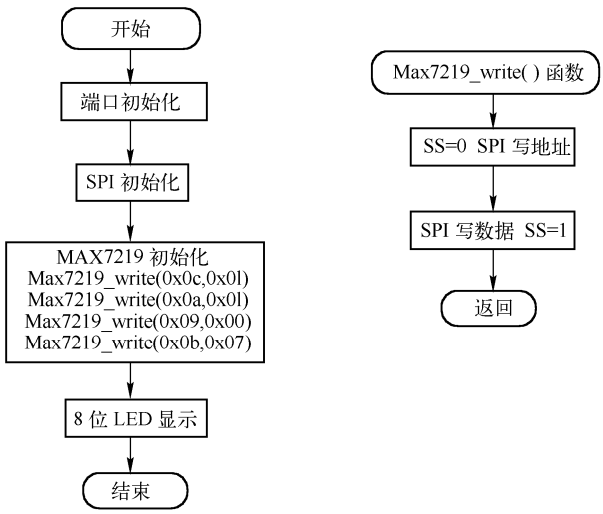


图 12-22　MAX7219 驱动数码管动态显示程序流程图

源程序

```
/*****************************************************
  文件名:          MAX7219 驱动数码管动态显示.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define ulong unsigned long
uchar LED_code[10]={0x30,0x6d,0x5b,0x30,0x7f,0x5f,0x6d,0x79};//12518623
//{0x7e,0x30,0x6d,0x79,0x33,0x5b,0x5f,0x70,0x7f,0x7b};      //表示不译方式 0～9
//B 码方式数字 0～9 时,uchar tab[10]={0,1,2,3,4,05,6,7,8,9};
void SPI_MasterTransmit(uchar i)                           //SPI 数据发送
{
   SPDR = i;                                               //启动数据串行输出
```

```
   while(!(SPSR==0x80))          //SPIF 等待传输结束
      {
        ;
      }
}
void Max7219_write(uchar address,uchar data)
{
   PORTB.4=0;
   SPI_MasterTransmit(address);
   SPI_MasterTransmit(data);
   PORTB.4=1;
}
void display(uchar *p)           //数码管 8 位显示 0～7
{
   uchar i;
   for(i=0;i<8;i++)
{
   Max7219_write(i+1,*(p+i));
}
}
void SPI_MasterInit(void)        //SPI 总线发送初始化
 {
    SPCR=0x53;                   //SPE=1，MSTR=1，SPR1=1，SPR0=1，使能 SPI 主机模式，
                                 //设置时钟速率为 fck/128
 }
void port_init(void)             //端口初始化
{
     DDRB=0xFF;
    PORTB=0xF0;
}
void   Max7219_init(void)
{
     Max7219_write(0x0c,0x01);       //0x0c 为关断模式寄存器；0x01 表示显示器处于工作状态
     Max7219_write(0x0a,0x0f);       //0x0a 为亮度调节寄存器；0x0f 使数码管显示亮度为最亮
     Max7219_write(0x09,0x00);       //0x09 为译码模式选择寄存器；0x00 为非译码方式;
                                     //0xff 选择为 8 位 BCD−B 译码模式
     Max7219_write(0x0b,0x07);       //0x0b 为扫描限制寄存器；0x07 表示可将 8 个 LED 数码管
}
void main(void)
{
   port_init();
   SPI_MasterInit();
   Max7219_init();
   while(1)
      {
        display(LED_code);
      }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“MAX7219 驱动数码管动态显示.COFF”文件，然后在 Proteus ISIS 中打开已创建的 MAX7219 驱动数码管动态显示电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试状态下，8 位共阴极 LED 数码显示“12518623”，仿真效果如图 12-23 所示。

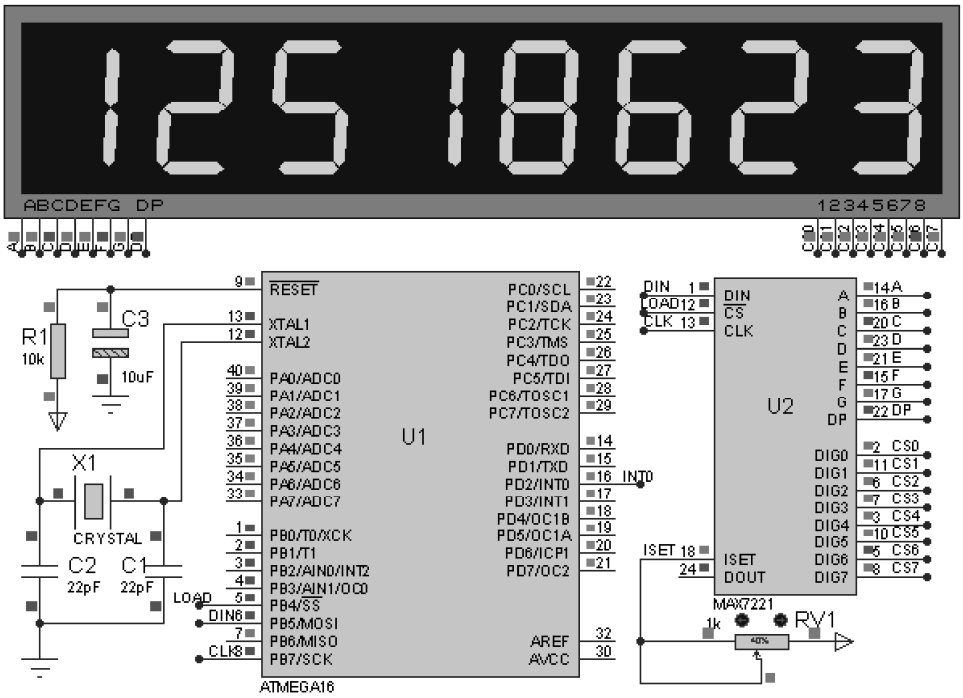


图 12-23　MAX7219 驱动数码管动态显示仿真效果图

项目十三　TWI 串行总线的应用

〖知识目标〗
☺ 了解 ATmega16 内部 TWI 总线接口结构、工作模式及相关的控制寄存器定义。
☺ 了解 ATmega16 内部 TWI 总线在单片机应用系统中的使用方法。
〖能力目标〗
☺ 能正确使用 ATmega16 内部 TWI 总线与外部元器件的连接。
☺ 掌握单片机对 TWI 串行总线器件的控制与编程。

I^2C（Inter-Integrated Circuit）总线是由 Philips 公司推出的一种两线式串行总线，是用于连接微控制器及其外围设备，实现同步双向串行数据传输的技术。I^2C 总线于 20 世纪 80 年代推出，是一种具有两线（串行数据线和串行时钟线）的标准总线。I^2C 总线的推出为单片机应用系统的设计带来了极大的方便，有利于系统设计的标准和模块化，减少了各个 PCB 之间的大量连线，从而提高了可靠性，降低了成本，使系统的扩展更加方便灵活。而 TWI（Two-Wire Serial Interface）总线是一种两线式的串行总线，它是 I^2C 总线的继承和发展。



任务 1　TWI 总线基础知识

AVR 系列单片机内部集成了 TWI 总线，该总线不仅全面兼容 I^2C 总线的特点，而且在操作和使用上比 I^2C 总线更为灵活，功能更加强大。与 SPI 总线和 USART 相比，I^2C 总线在通信时对时序的要求更加严格，协议也较复杂，因此在学习、使用 TWI 时，有必要了解 I^2C 总线的相关知识、TWI 模块结构及相关寄存器。

1．I^2C 总线特点

I^2C 总线有两条线，一条为 SDA 串行数据线，另一条为 SCL 串行时钟线。它采用“纯软件”的寻址方法，以减少连线数目。

I^2C 总线可工作在主/从方式。与 I^2C 总线相连的每个器件均对应一个特定的地址，可由芯片内部硬件和外部地址同时确定；每个器件在通信过程中可建立简单的主从关系，即主控器件可以作为发生器，也可以作为接收器。

I^2C 总线是一种真正的多主串行总线。多主器件竞争总线时，时钟同步和总线仲裁都由硬件自动完成。因为 I^2C 总线具有错误检测和总线仲裁功能，可以防止多个主控器件同时启动数据传输而产生的总线竞争。各个主机之间没有优先次序之分，也无中心主机。

串行数据在主从之间可以双向传输，其传输速率在不同的模式下各不相同，在标准模式下速率可达 100kbit/s，快速模式下可达 400kbit/s，高速模式下可达 3.4Mbit/s。

数据总线上的毛刺波由芯片上的滤波器滤除，确保数据的完整性。

同步时钟和数据线相配合，产生可以作为启动、应答、停止或重启动串行发送的握手信号。连接到同一总线的 I^2C 器件数只受总线的最大电容 400pF 限制。

2．I^2C 总线结构

I^2C 总线为双向同步串行总线，I^2C 设备与 I^2C 总线连接的接口电路如图 13-1 所示。

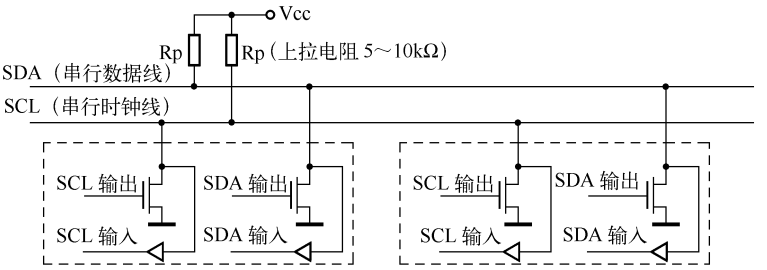
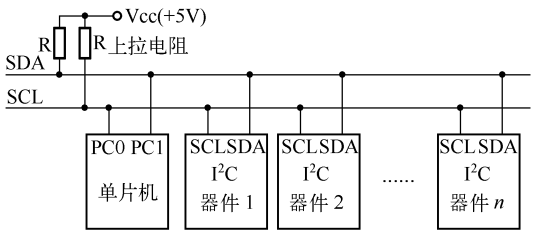


图 13-1　I^2C 设备与 I^2C 总线的接口电路

为了实现时钟同步和总线仲裁机制，应将电路上的所有输出连接，形成逻辑“线与”的关系。在进行时钟同步和总线仲裁机制过程中，I^2C 总线上的输出波形不是由哪个器件单独决定的，而是由连接在总线上所有器件的输出级共同决定的。由于 I^2C 总线端口为 FET 开漏输出结构，因此 I^2C 总线上必须接上拉电阻 Rp，Rp 的电阻值可参考有关数据手册来选择。总线在工作时，当 SDA 输出（或 SCL 输出），则 FET 管截止，输出为 0，带有 I^2C 接口的器件通过 SDA 输入（或 SCL 输入）的输入缓冲器采集总线上的数据或时钟信号。

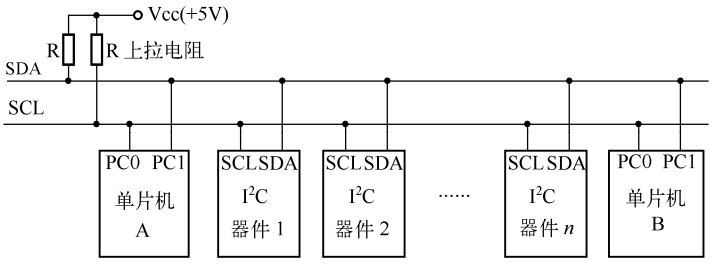
数据线 SDA 和时钟 SCL 构成的 I^2C 串行总线可发送和接收数据。I^2C 总线上发送数据的设备称为发送器，而接收数据的设备称为接收设备。能够初始发送、产生时钟启动/停止信号的设备称为主设备；被主设备寻址的设备称为从设备。在信息的传输过程中，I^2C 总线上并接的每个 I^2C 设备既是主设备（或从设备），又是发送器（或接收设备），这取决于它所要完成的功能。在标准模式下，单片机与 I^2C 设备之间、I^2C 设备与 I^2C 设备之间进行双向数据的传送，最高传送速率达 100kbit/s。单片机发出的控制信号分为地址码和控制量两部分，地址码用于选址，即接通需要控制的电路，确定控制的种类；控制量决定该调整的类别及需要调整的量。由于地址码和控制量的不同，各控制电路虽然处在同一条总线上，却彼此独立，互不相关。

I^2C 总线支持主/从和多主的两种工作方式，图 13-2（a）所示为主/从式系统的结构图；图 13-2（b）所示为多主式系统的结构图。



（a）主/从式系统结构

图 13-2　I^2C 总线系统的结构图



（b）多主式系统结构

图 13-2　I^2C 总线系统的结构图（续）

在图 13-15 中，若单片机不具有 I^2C 接口，可以利用单片机的口线模拟 SDA 和 SCL 线；若单片机本身具有 I^2C 接口，则可直接采用它的 SDA 和 SCL 口线。

3．I^2C 信号时序分析

【SDA 与 SCL 的时序关系】 I^2C 总线上的各位时序信号应符合 I^2C 总线协议，其时序关系如图 13-3 所示。整个串行数据与芯片本身的数据操作格式应相符。I^2C 总线为同步传输总线，总线数据与时钟完全同步。当时钟 SCL 线为高电平时，对应数据线 SDA 线上的电平即为有效数据（高电平为 “1”，低电平为 “0”）；当 SCL 线为低电平时，SDA 线上的电平允许改变；当 SCL 发出的重复时钟脉冲每次为高电平时，SDA 线上对应的电平就是逐位地传送数据，最先传输的是字节的最高位数据。

【起始与停止信号】I^2C 总线在传送数据时，有两种时序信号，即起始信号和停止信号，如图 13-4 所示。

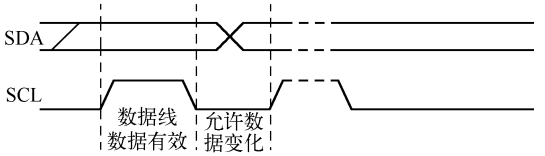


图 13-3　SDA 与 SCL 的时序关系

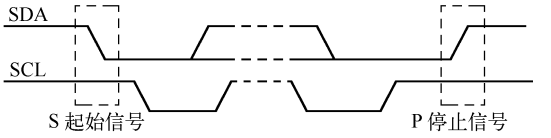


图 13-4　I^2C 总线的启动与停止信号

☺ S 启动信号（Start Condition）：当 SCL 为高电平时，SDA 出现由高到低的电平跳变为启动信号 S，由它启动 I^2C 总线的传送。

☺ P 停止信号（Stop Condition）：当 SCL 为高电平时，SDA 出现由低到高的跳变为结束信号 P，停止 I^2C 总线的数据传送。停止信号可将 E^2PROM 置于低功耗和备用方式（Stand by Mode）。

启动信号和停止信号都是由主设备产生的。在总线上的 I^2C 设备能很快地检测到这些信号。

【应答信号 ACK（Acknowledgement）和非应答信号 $\overline{\text{ACK}}$】 在 I^2C 总线上，所有的数据都是 8 位传送的，每个字节传送完后，都要有一个应答位，而第 9 个 SCL 时钟对应于应答信号位。当 SDA 线上为低电平时，第 9 个 SCL 时钟对应的数据位为应答信号 ACK；当 SDA 线上为高电平时，第 9 个 SCL 时钟对应的数据位为非应答信号 $\overline{\text{ACK}}$。此信号是由接收数据的设备发出的，如图 13-5 所示。

【数据传输】 当 I^2C 总线启动后或应答信号后的 1~8 个时钟脉冲，各对应一个字节的

D_7~D_0 位数据。在 SCL 时钟脉冲高电平期间，SDA 的电平必须保持稳定不变的状态，只有当 SCL 处在低电平时，才可以改变 SDA 的电平值，但起始信号和停止信号是特例。因此，当 SCL 处于高电平时，SDA 的任何跳变都会识别成为一个起始或停止信号。在数据传输过程中，发送 SDA 信号线上的数据以字节为单位，每个字节必须为 8 位，而且都是高位在前，低位在后，每次发送数据字节数量不受限制。数据传输如图 13-6 所示。

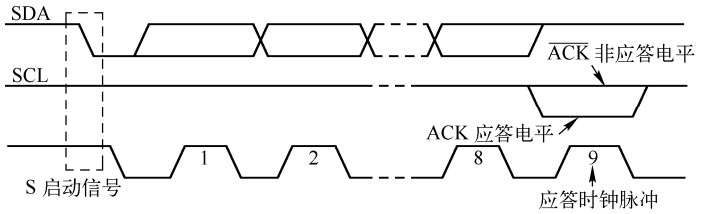


图 13-5　应答信号和非应答信号

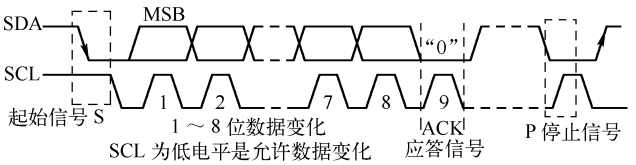


图 13-6　数据传输

【时钟同步】 所有器件在 SCL 线上产生自己的时钟来传输 I^2C 报文，这些数据只有在 SCL 高电平期间才有效。假设总线上有两个主设备，这两个主设备的时钟分别为 CLK1 和 CLK2。若在某一时刻这两个主设备处在不同的时钟脉冲，如图 13-7 所示，此时它们都想控制总线，让自己的数据进行传输，这时总线通过“线与”的逻辑关系来裁定有效时钟，产生时钟同步信号 SCL。

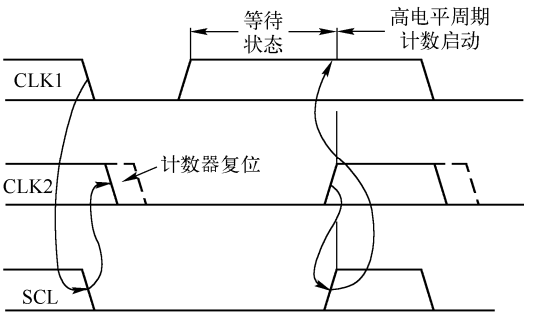


图 13-7　时钟同步

【仲裁】 数据在总线空闲时才能进行传输。在那些只有一个主设备的基本系统中是不会有仲裁的。然而，更多的复杂系统允许有多个主控设备，因此就有必要用某种形式的仲裁来避免总线冲突和数据丢失。通过用“线与”连接 I^2C 总线的两路信号（数据和时钟）可以实现仲裁。所有的主设备必须监视 I^2C 的数据和时钟线，如果主设备发现已经有数据正在传输，它就不会开始进行另一数据的传输。假设总线上有两个主设备，这两个设备的数据输出端分别对应为 DATA1 和 DATA2。这两个主设备可能在最小持续时间内产生一个起始条件，

都想获得控制总线的能力，向总线发出一个启动信号。在这种情况下，相互竞争的设备自动使它们的时钟保持同步，然后像平常一样继续发射信号。因为这是符合规定的起始条件，但总线不能同时响应这两个启动信号。鉴于此，I^2C 总线通过“线与”的逻辑关系产生在数据线 SDA 上的信号，如图 13-8 所示。

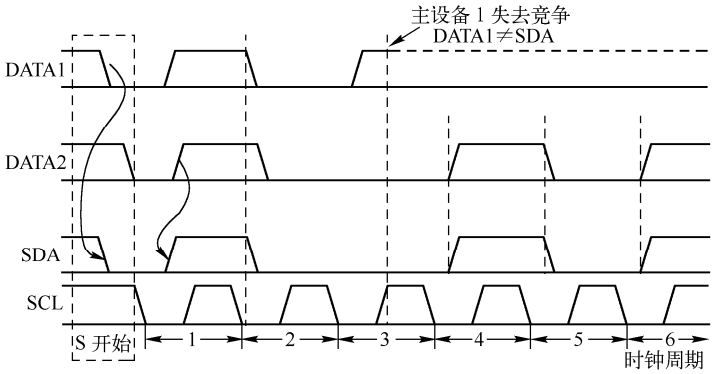


图 13-8　总线仲裁

从图中可以看出，刚开始 DATA1、DATA2 同时为高电平时，SDA 为高电平；DATA1 由高变低，而 DATA2 高电平保持时，SDA 变为低电平；DATA1、DATA2 同时为低电平时，SDA 为低电平；DATA1、DATA2 同时由低变为高电平时，SDA 也变为高电平；DATA1 为高电平，DATA2 为低电平时，SDA 为低电平，之后当 DATA2 由低变为高时，SDA 才为高……整个的过程就是 SDA=DATA1&DATA2，即 SDA 是 DATA1 与 DATA2“线与”的结果，之后，DATA2 获得了总线控制权，可以在总线上进行数据的传输，从而实现了总线仲裁。

因为没有数据丢失，仲裁处理是不需要一种特殊的仲裁相位的。获得主控权的设备从本质上来说，是不知道它曾为了总线而和其他设备竞争的。在串行数据传输中，若有重复起始条件或停止条件发送到总线上时，总线仲裁继续进行，不会停止。

4．I^2C 总线串行传输格式

I^2C 总线在串行传输数据中，启动后每次传送一个字节，每字节 8 位，且高位（D7）在前。其传输格式是，由主设备发出起始信号 S 启动 I^2C 总线，首先发出一个地址字节，此字节的高 7 位 SLAVE 作为从设备地址，最低位是数据的传送方向位，用 $R/\overline{W}$ 表示，即读/写选择位。$R/\overline{W}$ 位=0 时，表示主设备向从设备发送数据（即主设备将数据写入从设备）；$R/\overline{W}$ 位=1 时，表示发送器地址的主设备接收从设备发来的数据（即从设备向主设备发送数据）。在 SAVE+$R/\overline{W}$ 地址字节之后，发送器可发出任意个字节数据。每发送一个字节后，从设备都会做出响应，回送 ACK 应答信号。主设备收到 ACK 信号后，可继续发送下一个字节数据。如果从设备正在处理一个实时事件而不能接收主设备发来的字节时，如从设备正在处理一个内部中断，在这个中断处理前就不能接收主设备发给它的字节，可以使时钟 SCL 线保持为低电平，从设备必须使 SDA 保持高电平。此时主设备发出结束信号 P，使传送异常结束，迫使主设备处理等待状态。若从设备处理完毕时将释放 SCL 线，主器件将继续传送字节。连续传送数据的格式如下：

S	SLAVE	R/$\overline{W}$	A	DATA1	A	DATA2	A……DATAn	A/$\overline{A}$	P
1 位	7 位	1 位	1 位（ACK）	8 位	…	…	…	1 位	1 位结束
起始	器件地址字节	应答	8 位数据	…	…	…			结束

5．TWI 模块结构

AVR 的 TWI 模块由总线接口单元、比特率发生器、地址匹配单元和控制单元等部分组成，如图 13-9 所示。图中位于粗线中的寄存器都可以通过 AVR 数据总路线进行访问。

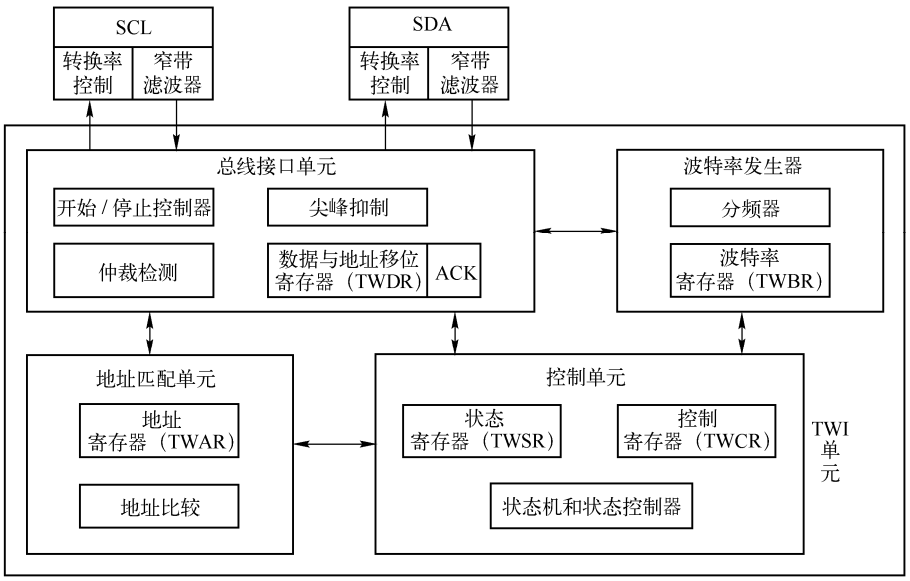


图 13-9　AVR 单片机的 TWI 模块结构

【SCL 的 SDA 引脚】 SCL 与 SDA 为 MCU 的 TWI 接口引脚。引脚的输出驱动器包含一个波形斜率限制器，以满足 TWI 规范。引脚的输入部分包括尖峰抑制单元，以去除小于 50ns 的毛刺。当相应的端口设置为 SCL 与 SDA 引脚时，可以使能 I/O 端口内部的上拉电阻，这样可省掉外部的上拉电阻。

【波特率发生器】 TWI 工作于主机模式时，波特率发生器控制时钟信号 SCL 的周期，并驱动时钟线 SCL。时钟 SCL 的周期由 TWI 状态寄存器 TWSR 的预分频系数及波特率寄存器 TWBR 设定。当 TWI 工作在从机模式时，不需要对波特率或预分频进行设定，但从机的 CPU 时钟频率必须大于 TWI 时钟线 SCL 频率的 16 倍。

📖 注意：

从机可能会延长 SCL 低电平的时间，从而降低 TWI 总线的平均时钟周期。SCL 的频率根据以下的公式产生：

$$f_{\mathrm{SCL}}=\frac{f_{\mathrm{CPUClock}}}{16+2(\mathrm{TWBR})\times 4^{\mathrm{TWPS}}}$$

式中，TWBR 为 TWI 波特率寄存器的值；TWPS 为 TWI 状态寄存器预分频位的值。在主机模式下，TWBR 的值为应大于 10，否则可能会产生不正确的输出。

【总线接口单元】 该单元包括数据与地址移位寄存器 TWDR，START/STOP 控制器

（开始/停止）和总线仲裁判定硬件电路。TWDR 寄存器用于存放发送或接收的数据或地址。除了 8 位的 TWDR 外，总线接口单元还有一个用于发送或接收应答的（N）ACK 寄存器。这个（N）ACK 寄存器不能由程序直接读/写。当接收数据时，它可以通过 TWI 控制寄存器 TWCR 来置位或清零；在发送数据时，（N）ACK 值由 TWCR 的设置决定。

START/STOP 控制器负责产生和检测 TWI 总线上的 START（启动）、REPEATED START（重新启动）与 STOP（停止）状态。即使在 MCU 处于休眠状态时，START/STOP 控制器仍然能够检测 TWI 总线上的 START/STOP 条件，当检测到 MCU 被 TWI 总线上的主机寻址访问时，将 MCU 从休眠状态唤醒。

如果 TWI 以主机模式启动了数据传输，仲裁检测电路将持续监听总线，以确定是否可以通过仲裁获得总线控制权。如果总线仲裁单元检测到自己在总线仲裁中丢失了总线控制权，则通知 TWI 控制单元执行正确的动作，并产生合适的状态码。

【地址匹配单元】 地址匹配单元将检测从总线上接收到的地址是否与 TWAR 寄存器中的 7 位地址相匹配。如果 TWAR 寄存器的 TWI 广播应答识别允许位 TWGCE 为“1”，则从总线接收到的地址也会与广播地址进行比较。一旦地址匹配成功，控制单元将得到通知以进行正确的响应。TWI 可以响应，也可以不响应主机的寻址，这取决于 TWCR 寄存器的设置。即使 MCU 处于休眠状态时，地址匹配单元仍可继续工作。一旦主机寻址到这个器件，就可以将 MCU 从休眠状态唤醒。

【控制单元】 控制单元监听 TWI 总线，并根据 TWI 控制寄存器 TWCR 的设置做出相应的响应。当 TWI 总线上产生需要应用程序干预处理的事件时，TWI 中断标志位 TWINT 置位。在下一个时钟周期，将表示这个事件的状态字写入 TWI 状态寄存器 TWSR 中。在其他时间里，TWSR 的内容为一个表示无事件发生的特殊状态字。一旦 TWINT 标志位置“1”，时钟线 SCL 即被拉低，暂停 TWI 总线上的数据传输，让用户程序处理事件。

在下列状况出现时，TWINT 标志位置位：在 TWI 传送完 START/REPEATED START 信号之后；在 TWI 传送完 SLA+R/W 数据之后；在 TWI 传送完地址字节之后；在 TWI 总线仲裁失败之后；在 TWI 被主机寻址之后（广播方式或从机地址匹配）；在 TWI 接收到一个数据字节之后；作为从机工作时，TWI 接收到 STOP 或 REPEATED START 信号之后；由于非法的 START 或 STOP 信号造成总线错误时。

6．TWI 相关寄存器

【TWI 波特率寄存器 TWBR】

Bit	7	6	5	4	3	2	1	0
	TWBR7	TWBR6	TWBR5	TWBR4	TWBR3	TWBR2	TWBR1	TWBR0
读 / 写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

☺ TWBR7~TWBR0（bit7~bit0）：TWI 波特率寄存器位。TWI 工作于主机模式时，波特率发生器控制时钟信号 SCL 的周期，由 TWBR 的状态寄存器 TWSR 的预分频系数决定；当 TWI 工作于从机模式时，不需要设置波特率寄存器或预分频系数，但从机的 CPU 频率必须大于 TWI 时钟线 SCL 频率的 16 倍。在主机模式下，SCL 频率可使用下式计算：

$$f_{\mathrm{SCL}}=\frac{f_{\mathrm{CPUClock}}}{16+2(\mathrm{TWBR})\times 4^{\mathrm{TWPS}}}$$

【TWI 控制寄存器 TWCR】 TWCR 用于控制 TWI 操作。它可用于使能 TWI，通过施加 START 到总线上来启动主机访问、产生接收器应答、产生 STOP 状态，以及在写入数据到 TWDR 寄存器时控制总线的暂停等。在禁止访问 TWDR 期间，试图将数据写入到 TWDR 时，要给出写入冲突信息。

Bit	7	6	5	4	3	2	1	0
	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	–	TWIE
读/写	R/W	R/W	R/W	R/W	R	R/W	R	R/W
初始值	0	0	0	0	0	0	0	0

☺ TWINT（bit7）：TWI 中断标志位。当 TWI 完成当前工作，希望应用程序响应时 TWINT 置“1”。若 SREG 的 I 标志及 TWCR 寄存器的 TWIE 标志也置“1”，则 MCU 执行 TWI 中断服务程序。当 TWINT 置“1”时，SCL 信号的低电平被延长。对 TWINT 标志位清“0”时，必须通过软件写“1”来完成。执行中断时，硬件不会自动将其改写为“0”。

> 注意：
> 只要该位被清“0”，TWI 立即开始工作。因此，在清零 TWINT 前一定要首先完成对地址寄存器 TWAR、状态寄存器 TWSR 及数据寄存器 TWDR 的访问。

☺ TWEV（bit6）：TWI 应答（ACK）允许位。TWEA 位控制应答 ACK 信号的发生。如果 TWEA=1，出现以下条件时接口发生 ACK 脉冲：①器件的从机地址与主机发出的地址相符合；②TWAR 的 TWGCE 置位时接收到广播呼叫；③在主机/从机接收模式下接收到一个字节的数据。如果 TWEA=0，器件暂时虚拟地脱离 TWI 总线。TWEV 置“1”后，器件重新恢复地址识别。

☺ TWSTA（bit5）：TWI 启动（START）信号状态位。若要将器件设置为串行总线上的主机时，TWSTA 应置“1”。TWI 硬件检测总线是否空闲，如果总线处于空闲时，则将在总线上产生启动（START）信号；如果总线并不空闲，则 TWI 将等到总线上一个停止（STOP）信号被检测到后，再发出一个新的启动信号，以获得总线的控制权而成为主机。发送 START 后，TWSTA 位必须软件清“0”。

☺ TWSTO（bit4）：TWI 停止（STOP）信号状态位。在主机模式下，如果 TWSTO 置“1”，TWI 接口将在总线上产生 STOP 信号，然后 TWSTO 自动清零；在从机模式下，如果 TWSTO 置“1”，可以使接口从错误状态恢复到未被寻址的状态，此时总线上不会有 STOP 状态产生，但 TWI 返回一个定义好且未被寻址的从机模式，并释放 SCL 和 SDA 线为高阻态。

☺ TWWC（bit3）：TWI 写冲突标志位。当 TWINT=0 时，写数据寄存 TWDR 将置位 TWWC；当 TWINT=1 时，每一次对 TWDR 的写访问都将更新此标志。

☺ TWEN（bit2）：TWI 允许位。TWEN=1 时，TWI 引脚将 I/O 引脚切换到 SCL 与 SDA 引脚，使能波形斜率限制器和毛刺滤波器。TWEN=0 时，TWI 接口模块将被关闭，所有 TWI 传输将被终止。

☺ TWIE（bit0）：TWI 中断使能位。该位写为“1”且 SREG 寄存器的 I 位置位时，只要 TWINT 标志位为“1”，TWI 中断请求就使能。

【TWI 状态寄存器 TWSR】

Bit	7	6	5	4	3	2	1	0
	TWS7	TWS6	TWS5	TWS4	TWS3	—	TWPS1	TWPS0
读/写	R	R	R	R	R	R	R/W	R/W
初始值	1	1	1	1	1	0	0	0

☺ TWS7~TWS3（bit7~bit3）：TWI 状态位。这 5 位用于反映 TWI 逻辑和总线的状态，从 TWSR 读出的值包括 5 位状态值与 2 位预分频值。检测状态位时，设计者应屏蔽预分频位，使它为“0”。这使状态检测独立于预分频器设置。

☺ TWPS1、TWPS0（bit1、bit0）：TWI 预分频位。这两位可读/写，用于控制波特率预分频的值，见表 13-1。

表 13-1　TWI 波特率预分频器

TWPS1	TWPS0	预分频器值	TWPS1	TWPS0	预分频器值
0	0	1	1	0	16
0	1	4	1	1	64

【TWI 数据与移位寄存器 TWDR】 在发送模式下，TWDR 包含了要发送的字节；在接收模式，TWDR 包含了接收到的数据。当 TWI 接口没有进行移位工作（TWINT 置位）时，该寄存器可以被写入。第一次中断发生前，用户不能够初始化数据寄存器。若 TWINT 置“1”，TWDR 中的数据将保持稳定。在数据移出时，总线上的数据同时移入寄存器。因此，TWDR 的内容总是包含了总线上出现的最后一个字节，除非 MCU 是从掉电或省电模式被 TWI 中断唤醒。总线仲裁失败时，主机将切换为从机，但总线上出现的数据不会丢失。TWI 硬件逻辑电路自动控制 ACK 的处理，CPU 不能直接访问 ACK。

Bit	7	6	5	4	3	2	1	0
	TWD7	TWD6	TWD5	TWD4	TWD3	TWD2	TWD1	TWD0
读 / 写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	1	1	1	1	1	1	1	1

☺ TWD7~TWD0（bit7~bit0）：TWI 数据与移位寄存器位。这 8 位包括将要传送的下一个数据字节，或者 TWI 总线上最后接收到的一个数据字节。

【TWI 地址寄存器 TWAR】 TWAR 地址寄存器的高 7 位为从机地址。当 ATmega16 的 TWI 工作于从机模式时，在 TWAR 中应设置从机的寻址地址，TWI 将根据这个地址进行响应。当 ATmega16 的 TWI 工作于主机模式时，不需要此地址。在多主机系统中，如果器件既可以作为主机，又可以作为从机时，则必须 TWAR 寄存器。TWAR 的 LSB（最低位）用于识别通用地址（或广播地址 0x00）的识别允许位。在器件内有一个地址比较器，一旦接收到的地址和本机地址一致，则产生 TWI 中断请求。

Bit	7	6	5	4	3	2	1	0
	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE
读 / 写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	1	1	1	1	1	1	1	0

☺ TWA6~TWA0（bit7~bit1）：TWI 从机地址寄存器位。这 7 位用于存放 TWI 单元的从机地址。

☺ TWGCE（bit0）：TWI 通用呼叫（或广播呼叫）识别允许位。如果该位置“1”，则将使能对 TWI 总线上通用地址（或广播呼叫）的呼叫和识别。

7. I^2C 总线函数

在 CodeVisionAVR 编译器中提供了表 13-2 所列的 I^2C 总线操作基本函数，利用这些函

数可以非常快地实现 AVR 与 I^2C 外围器件之间的数据读/写操作。

表 13-2 CodeVisionAVR 提供的 I^2C 总线函数

函 数 名	功 能 描 述
void i2c_init(void)	初始化 I^2C 总线，调用其他 I^2C 函数前，必须先调用此函数
unsigned char i2c_start(void)	发送 START 信号，如果总线空闲，则返回“1”；如果总线忙，则返回“0”
void i2c_stop(void)	发送 STOP 信号
unsigned char i2c_read(unsigned char ack)	读 1 字节，参数定义了在 1 字节被读取后是否要发送请求，可以置 ACK 为 0 或 1
unsigned char i2c_write(unsigned char ack)	写 1 字节，如果从机有应答，则返回“1”；如果从机不应答，则返回“0”

任务 2 24C256 开启次数读/写控制

使用 FM24C256 存储器统计 ATmega16 单片机的开启次数，要求单片机开启后就显示开启的次数。当开启的次数小于或等于 5 次时，绿色 LED 点亮；当次数大于 5 次且小于等于 10 次时，红色 LED 点亮；如果开启次数超过 10，FM24C256 清零并重新开始统计开启次数；数码管由 74HC595 采用 SPI 总线驱动。

〖**FM24C256 基本知识**〗 FM24C256 是采用先进的铁电技术制造的 256kbit 的非易失性存储器。铁电随机存储器（Ferroelectric Random Access Memory，FRAM）具有非易失性，并且可以像 RAM 一样快速读/写，数据在掉电后可以保存 10 年。相对于 E^2PROM 或其他非易失性存储器，FRAM 具有系统可靠性更高，结构更简单等诸多优点。

FM24C256 以总线速度进行写操作，无须延时。在进行写操作时，下一个总线周期可以立刻开始而无须进行数据轮循。另外，FM24C256 的读/写操作次数比 E^2PROM 的读/写操作次数要多，可以承受超过 100 亿次的读/写操作，同时由于无须内部电压提升电路，FM24C256 具有非常低的写操作功耗。

【引脚功能】 FM24C256 使用工业标准两线接口，8 脚 SOP 封装，如图 13-10 所示。A_0、A_1、A_2 为器件地址配置；V_{SS} 为电源地；SDA 为 I^2C 总线接口数据线；SCL 为 I^2C 总线接口时钟线；WP 为写保护，高电平有效；VDD 为电源正极。

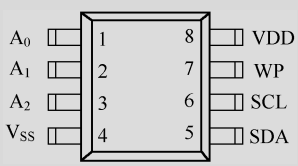


图 13-10 FM24C256 封装形式

【器件地址和片内存储器地址】 一般微处理器/微控制器称为主器件，它产生串行时钟和起始停止信号；E^2PROM 工作在从机方式，称为从器件。主器件和从器件都可以作为发送器或接收器，主器件控制数据的传输。通过片选或页面选择地址输入端 A_0、A_1、A_2 可以将 E^2PROM 从器件连接到总线上，即在多片组的情况下通过 A_0、A_1、A_2 进行芯片的寻址。

主器件发送“启动”信号后，再发送一个 8 位的含有芯片地址的控制字对从器件进行片选。这 8 位片选地址字由 3 部分组成：第 1 部分是 8 位控制字的高 4 位（D7~D4），固定为 1010 是 I^2C 总线器件特征编码；第 2 部分是最低位 D0，D0 位是读/写选择位 $R/\overline{W}$，决定微处理器对 E^2PROM 进行读/写操作，$R/\overline{W}=1$，表示读操作，$R/\overline{W}=0$ 表示写操作；剩下的 3 位为第 3 部分即 A_0、A_1、A_2，这 3 位是芯片选择位，它们必须和外

部地址脚信号相对应。

FM24C256 内部的存储器容量为 32KB，采用线性连续排列，地址空间为 0x0000~0x7FFF，因此 FM24C256 内部存储器的地址长度为 15bit，需要 2B 表示。

FM24C256 内部有一个 15bit 的地址指针寄存器，里面保存着当前存储器单元的地址。对 FM24C256 的读/写，就是对该地址指针所指向的当前存储器单元进行操作。对当前存储器单元进行读/写操作后，地址指针会自动加 1，指向下一个存储器单元。如果当前地址为 0x7FFF，若对它读或写操作后，地址指针将变为 0x0000。



硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 13-3 所列的元器件，并完成图 13-11 所示的硬件电路图设计。

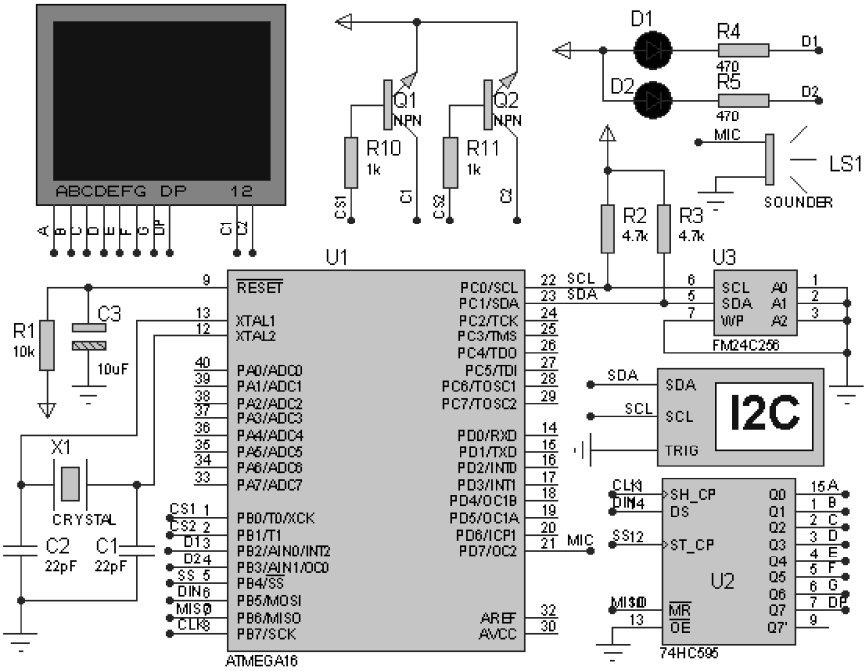


图 13-11　24C256 开启次数读/写控制电路图

表 13-3　24C256 开启次数读/写控制所使用的元器件

单片机 ATmega16	电解电容 CAP-ELEC 10uF	瓷片电容 CAP22pF	电阻 RES
晶振 CRYSTAL 8MHz	2 位共阳数码管 7SEG-MPX2-CA-BLUE	串行移位 74HC595	FM24C256
发光二极管 LED-GREEN	发光二极管 LED-RED	蜂鸣器 SOUND	三极管 NPN



软件设计

由于 CodeVisionAVR 中提供了 I^2C 总线函数，因此可使用这些函数来完成本设计。采用此方法时，先要声明使用单片机的哪些口线作为 I^2C 总线。要实现开启次数统计，每次单片机开启时，应先读取 FM24C256 中的数据，再将该数据加 1，并进行相关判断后存储到 FM24C256 中即可。

对 FM24C256 进行读操作时，其流程为 ATmega16 对其发送启动信号→发送 FM24C256 的从机地址→指定存储单元地址→重新发送启动信号→发送从机读寻址→读取数据→发送停止信号。对 FM24C256 进行写操作时，其流程为 ATmega16 对其发送启动信号→发送 FM24C256 的从机地址→指定存储单元地址→写数据→发送停止信号。本设计的程序流程图如图 13-12 所示。

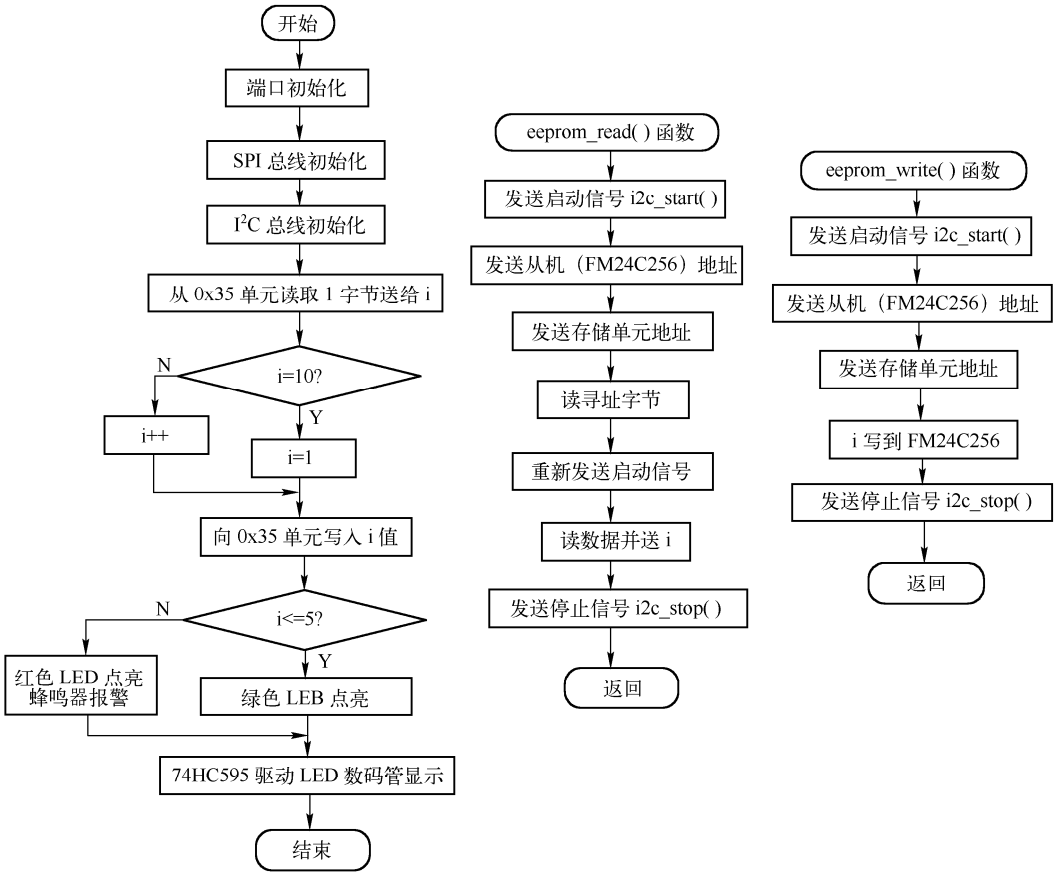


图 13-12　24C256 开启次数读/写控制程序流程图

使用 CodeVisionAVR 提供的 I²C 总线函数可以实现设计要求，但是应注意采用此方法时并没有使用 AVR 的 TWI 硬件接口。

源程序

```
/*****************************************************
  文件名:          24C256 开启次数读写控制.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#include "i2c.h"                                  //使用 CVAVR 的 IIC 函数
#include "delay.h"                                //使用 CVAVR 的延时函数
```

```
#define uchar unsigned char
#define uint unsigned int
#define ulong unsigned long
#define EEPROM_BUS_ADDRESS 0xa0
#asm
   .equ __i2c_port=0x15                    //使用 PORTC 端口作 I2C 总线
   .equ __sda_bit=1                        //SDA 使用 PC1
   .equ __scl_bit=0                        //SCL 使用 PC0
#endasm
#define   CS1   PORTB.0
#define   CS2   PORTB.1
#define   D1    PORTB.2
#define   D2    PORTB.3
#define   MIC   PORTD.7
uchar disp[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,0xBF};//0～9、
uint cnt,cnt_H,cnt_L;
bit    sp;
void sound(void)                           //报警声音
{
   uchar i;
   for(i=0;i<200;i++)
      {
         sp=～sp;
         MIC=sp;                           //输出频率 1kHz
         delay_us(500);                    //延时 500us
      }
   for(i=0;i<200;i++)
      {
         sp=～sp;
         MIC=sp;                           //输出频率 500Hz
         delay_us(500);                    //延时 1ms
         delay_us(500);
      }
}
void SPI_MasterTransmit(uchar i)           //SPI 数据发送
{
   SPDR = i;                               //启动数据串行输出
   while(!(SPSR==0x80))                    //SPIF 等待传输结束
      {
        ;
      }
}
void SPI_MasterInit(void)                  //SPI 总线发送初始化
 {
    SPCR=0x53;                  //SPE=1，MSTR=1，SPR1=1，SPR0=1，使能 SPI 主机模式，
                                //设置时钟速率为 fck/128
```

```
 }
void disp_595(uchar i)
{
   PORTB &= ～(1 << 4);                          //准备锁存
   SPI_MasterTransmit(i);
   PORTB |= (1 << 4);                            //锁存数据
}
uchar eeprom_read(uint address)
 {
    uchar data;
    i2c_start();                                 //发起始信号
    i2c_write(EEPROM_BUS_ADDRESS);               //发送从机写寻址字节
    i2c_write(address >>8 );                     //发送存储单元地址高字节
    i2c_write(address);                          //发送存储单元地址低字节
    i2c_start();                                 //发起始信号
    i2c_write(EEPROM_BUS_ADDRESS | 1);           //发送从机写寻址字节
    data=i2c_read(0);                            //读 1B 数据，返回 nACK
    i2c_stop();                                  //发停止信号
    return data;
 }
void eeprom_write(uint address,uchar data)
 {
    i2c_start();                                 //发起始信号
    i2c_write(EEPROM_BUS_ADDRESS);               //发送从机写寻址字节
    i2c_write(address>>8);                       //发送存储单元地址高字节
    i2c_write(address);                          //发送存储单元地址低字节
    i2c_write(data);                             //写 1B 数据到 24C256
    i2c_stop();                                  //发停止信号
    delay_ms(10);                                //延时 10ms 等待写操作完成
 }
void RD_WR(void)                                 //开机进行 24C256 读/写操作
{
   cnt=eeprom_read(0x35);                        //从地址 0x35 读一个字节
   if(cnt==10)
      {
        cnt=1;
      }
   else
      {
        cnt++;
      }
   eeprom_write(0x35,cnt);                       //在地址 0x35 中写入 cnt
}
void data_op(void)                               //24C256 开启次数操作
{
   if(cnt<=5)
```

```
        {
          D1=0;
          D2=1;
        }
    else if(cnt>5)
        {
          D1=1;
          D2=0;
          sound();
        }
}
void data_in(void)
{
   cnt_H=cnt/10;
   cnt_L=cnt%10;
}
void display(void)
{
   data_in();
   CS1=1;
   disp_595(disp[cnt_H]);
   delay_ms(8);
   CS1=0;
   CS2=1;
   disp_595(disp[cnt_L]);
   delay_ms(8);
   CS2=0;
}
void port_init(void)                    //端口初始化
{
     DDRB=0xFF;
    PORTB=0xF0;
     DDRD=0x80;
    PORTD=0x00;
}
void main(void)
{
   port_init();
   SPI_MasterInit();
   i2c_init();                          //初始化 I2C 总线
   RD_WR();
   while(1)
        {
          data_op();
          display();
        }
}
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“24C256 开启次数读/写控制.COFF”文件，然后在 Proteus ISIS 中打开已创建的 24C256 开启次数读/写控制电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试状态下，其运行效果如图 13-13 所示。再单击按钮，然后再单击按钮，LED 显示的次数为上次次数加 1。当显示次数在 5 次范围内时，绿色 LED 点亮；当显示到 5 次后，单击按钮，再按单击按钮时，红色 LED 点亮，并发出报警声；当显示达到 10 次后，单击按钮，再单击按钮时，LED 显示为 1，重新开始计数。从图 13-13 中还可以看出，在 I^2C 调试器工作界面中显示的数字与 LED 数码管显示的数据完全相同。

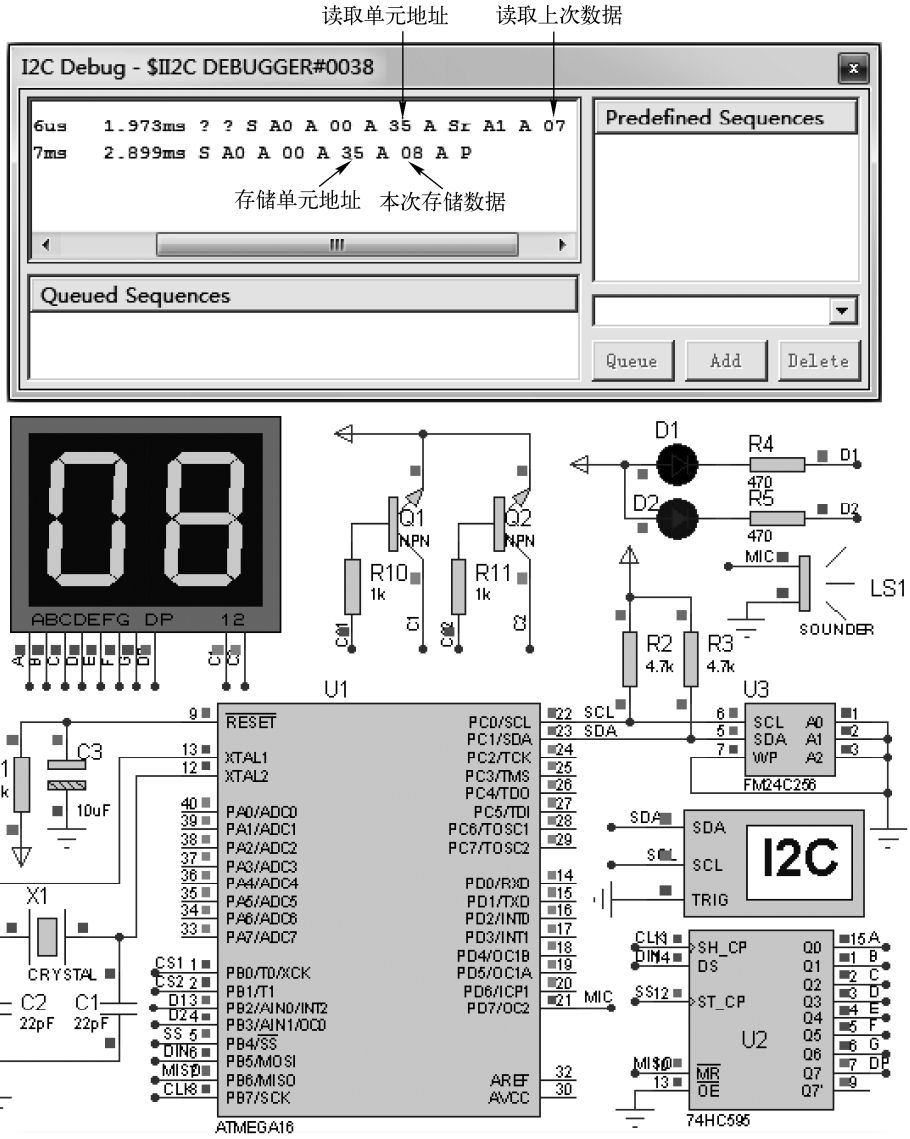


图 13-13　24C256 开启次数读/写控制仿真效果图

任务 3　PCF857 驱动 8 位数码管显示

使用 PCF8574 驱动 8 位数码管动态显示“12518623”，要求 PCF8574 由 CodeVision AVR 提供的 I^2C 总线函数控制。

〖**PCF8574 基础知识**〗 PCF8574 为 COMS 器件，它通过两条双向总线（I^2C）可使大多数单片机实现远程 I/O 口扩展。该器件包含一个 8 位准双向口和一个 I^2C 总线接口。PCF8574 电流消耗很低，且输出端口具有大电流驱动能力，可直接驱动 LED。它还带有一条中断接线（$\overline{INT}$）可与单片机的中断逻辑相连。通过 $\overline{INT}$ 发送中断信号，远程 I/O 口不必经过 I^2C 总线通信就可通知单片机是否有数据从端口输入。

【PCF8574 封装形式及引脚功能】 PCF8574 通常采用 DIL16、SO16、SSOP20 这 3 种封装形式，采用 SO16 封装形式的引脚配置如图 13-14 所示。各引脚功能如下所述。

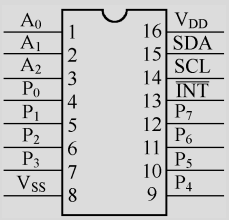


图 13-14　PCF8574 SO16 封装形式

☺ A_0、A_1、A_2：片选或选择地址输入。

☺ P_0~P_7：准双向 I/O 端口 1~7。

☺ V_{SS}：地线。

☺ $\overline{INT}$：中断输入，低电平有效。

☺ SCL：串行时钟输入端，用于输入/输出数据的同步。在其上升沿时串行写入数据，在下降沿时串行读取数据。

☺ SDA：串行数据（/地址）I/O 端口，用于串行数据的输入/输出。这个引脚是漏极开路驱动，可以与任何数量的漏极开路或集电极开路器件“线或”连接。

☺ V_{DD}：电源电压，接+5V。

【PCF8574 内部结构】 PCF8574 的内部结构如图 13-15 所示，它主要由准双向 I/O 端口、低通滤波器、中断逻辑、输入滤波器、I^2C 总线控制器、移位寄存器等部分组成。

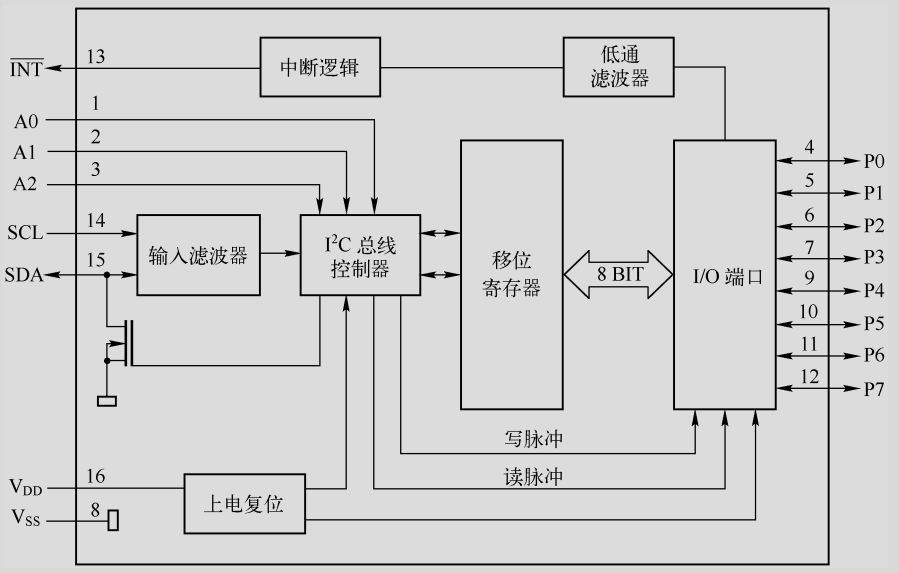


图 13-15　PCF8574 内部结构

【准双向 I/O 口】 PCF8574 的准双向 I/O 口可用做输入和输出，而不需要通过控制寄存器定义数据的方向。该模式中只有 V_{DD} 提供的电流有效，上电时，I/O 端口为高电平。在大负载输出时，提供额外的强上拉以使电平迅速上升。当输出写为高电平时，打开强上拉，在 SCL 的下降沿关闭上拉。I/O 端口用做输入端口前，I/O 应当为高电平。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 13-4 所列的元器件，并完成图 13-16 所示的硬件电路图设计。

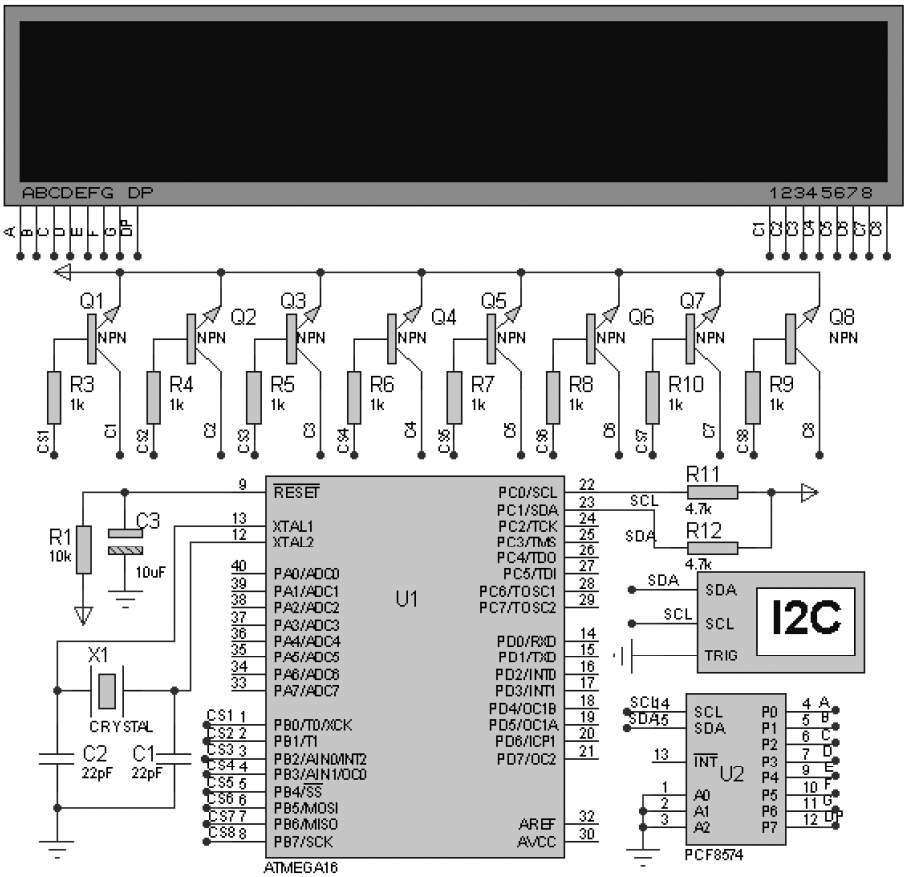


图 13-16　PCF8574 驱动 8 位数码管显示电路图

表 13-4　PCF857 驱动 8 位数码管显示所使用的元器件

单片机 ATmega16	电解电容 CAP-ELEC 10uF	瓷片电容 CAP22pF	电阻 RES
晶振 CRYSTAL 8MHz	8 位共阳数码管 7SEG-MPX8-CA-BLUE	串行扩展 PCF8574	三极管 NPN

软件设计

由于 CodeVisionAVR 中提供了 I^2C 总线函数，因此可使用这些函数来完成本设计。采

用此方法时，对 PCF857 进行写操作时，其流程为 ATmega16 对其发送启动信号→发送 PCF8574 的从机地址→写 1B 数据→发送停止信号。本设计的程序流程图如图 13-17 所示。

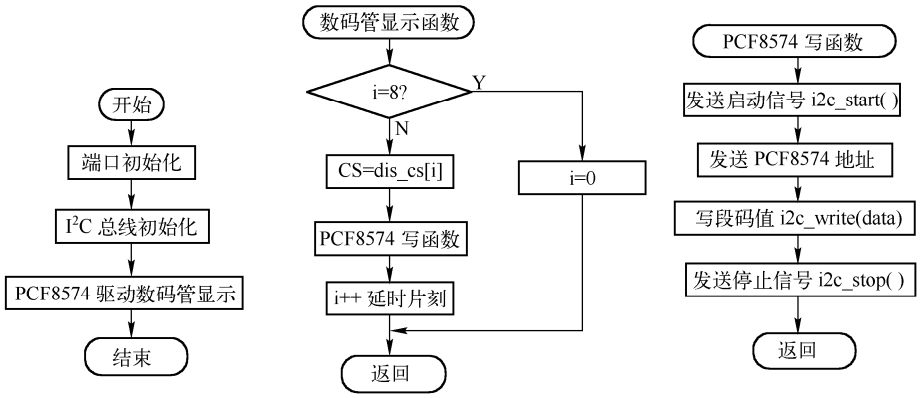


图 13-17　PCF8574 驱动 8 位数码管显示程序流程图

源程序

```
/*****************************************************
  文件名:          PCF8574 驱动 8 位数码管显示.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#include "i2c.h"                                    //使用 CVAVR 的 IIC 函数
#include "delay.h"                                  //使用 CVAVR 的延时函数
#define uchar unsigned char
#define uint unsigned int
#define ulong unsigned long
#define PCF8574_WR_ADDRESS 0x40
#asm
   .equ __i2c_port=0x15                             //使用 PORTC 端口作 I2C 总线
   .equ __sda_bit=1                                 //SDA 使用 PC1
   .equ __scl_bit=0                                 //SCL 使用 PC0
#endasm
#define   CS     PORTB
#define   CS1    PORTB.0
#define   CS2    PORTB.1
#define   CS3    PORTB.2
#define   CS4    PORTB.3
#define   CS5    PORTB.4
#define   CS6    PORTB.5
#define   CS7    PORTB.6
#define   CS8    PORTB.7
```

```
uchar disp_code[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,0xBF};  //0～9、
uchar disp_cs[]={0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80};       //定义 CS 片选代码
uchar disp_buff[]={1,2,5,1,8,6,2,3};                             //定义显示初值
void PCF8574_write(uchar data)
{
    i2c_start();                                                 //发起始信号
    i2c_write(PCF8574_WR_ADDRESS);                               //发送从机地址
    i2c_write(data);                                             //写 1B 数据到 PCF8574
    i2c_stop();                                                  //发停止信号
    delay_ms(3);                                                 //延时 3ms 等待写操作完成
}
void display(void)
{
  uint   i;
  CS=0x00;                                                       //CS 片选清零
  for(i=0;i<8;i++)
     {
       CS=disp_cs[i];                                            //选定相应 CS 值
       PCF8574_write(disp_code[disp_buff[i]]);                   //写段码值
       delay_ms(2);
     }
}
void port_init(void)                                             //端口初始化
{
     DDRB=0xFF;
   PORTB=0x00;
}
void main(void)
{
  port_init();
  i2c_init();                                                    //初始化 I2C 总线
  while(1)
     {
       display();
     }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“PCF8574 驱动 8 位数码管显示.COFF”文件，然后在 Proteus ISIS 中打开已创建的 PCF8574 驱动 8 位数码管显示电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试状态下，其运行效果如图 13-18 所示。

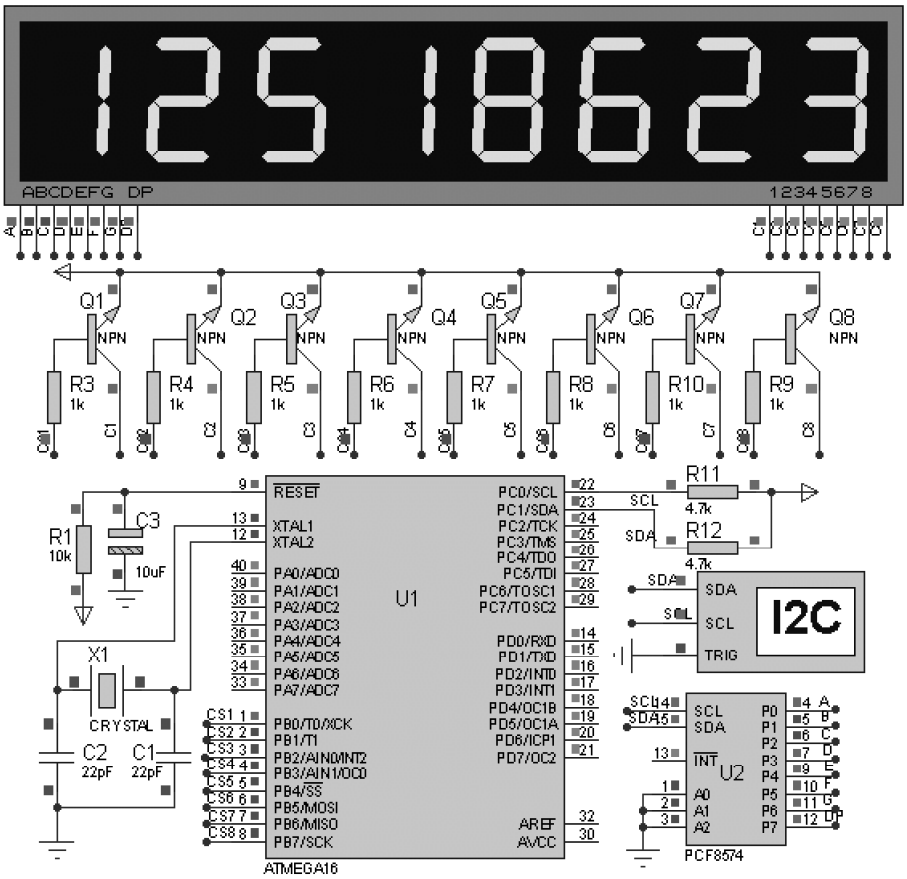


图 13-18　PCF8574 驱动 8 位数码管显示仿真效果图

项目十四　D/A 转换器的应用

〖知识目标〗

☺ 了解 D/A（数/模）转换器的基本知识;

☺ 了解并行和串行 D/A 转换器在单片机应用系统中的使用方法。

〖能力目标〗

☺ 能正确使用 D/A 转换器;

☺ 掌握单片机对 D/A 转换器的控制与编程。

任务 1　DAC0832 D/A 转换

使用单片机和 DAC0832 设计一个频率可调的正弦波信号发生器。

〖**DAC0832 基础知识**〗 DAC0832 是 8 位分辨率的并行数据输出 D/A 转换芯片，该芯片以其价格低廉、接口简单、转换控制容易等优点，在单片机应用系统中得到广泛的应用。

【DAC0832 外形及引脚功能】 DAC0832 是 20 引脚的双列直插式芯片，其外形如图 14-1 所示。各引脚功能如下所述。

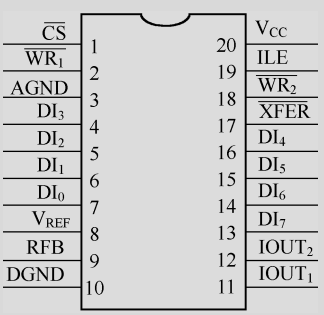


图 14-1　DAC0832 外形图

☺ $\overline{CS}$：片选信号，低电平有效。

☺ $\overline{WR_1}$：输入寄存器的写选通信号。

☺ AGND：模拟地，模拟信号和基准电源的参考地。

☺ DI_0~DI_7：数据输入线。

☺ V_{REF}：基准电压输入线（-10~+10V）。

☺ RFB：反馈信号输入线，芯片内部有反馈电阻。

☺ DGND：数字地。

☺ $IOUT_1$：电流输出线。当输入全为“1”时，$IOUT_1$ 最大。

☺ $IOUT_2$：电流输出线。其值与 $IOUT_1$ 之和为一常数。

☺ $\overline{XFER}$：数据传送控制信号输入线，低电平有效。

☺ $\overline{WR_2}$：DAC 寄存器写选通输入线。

☺ ILE：数据锁存允许控制信号输入线，高电平有效。

☺ V_{CC}：电源输入线（+5~+15V）。

【DAC0832 内部结构及工作原理】 DAC0832 的内部结构如图 14-2 所示。从图中可以看出 DAC0832 是双缓冲结构，有一个输入寄存器，一个 DAC 寄存器构成双缓冲结构。

DAC0832 是由 R-2R 的电阻阶梯网络来完成数字到模拟的转换，如图 14-3 所示。

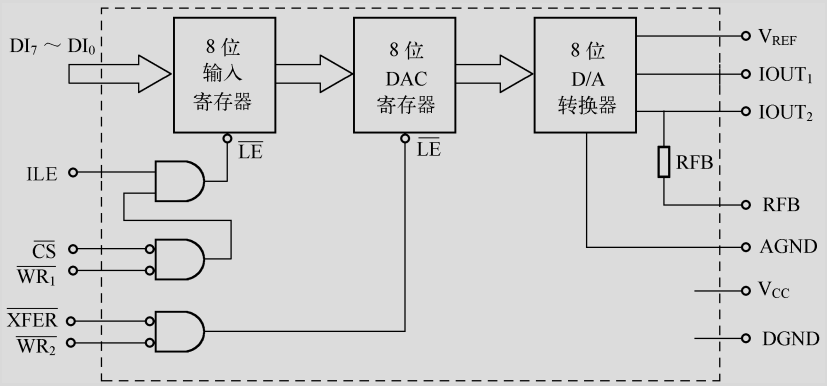


图 14-2　DAC0832 内部结构

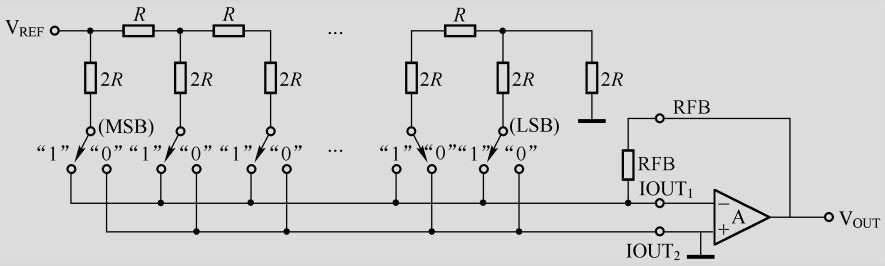


图 14-3　R-2R 电阻阶梯网络

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 14-1 所列的元器件，并完成图 14-4 所示的硬件电路图设计。

表 14-1　DAC0832D/A 转换所使用的元器件

单片机 ATmega16	电解电容 CAP-ELEC 10uF	瓷片电容 CAP22pF	电阻 RES
晶振 CRYSTAL 8MHz	数模转换 DAC0832	运放 OP1P	按钮 BUTTON

软件设计

在图 14-4 中，8 位 DAC0832 的输入数据与输出电压的关系为 $CHA = -\frac{VREF}{256} \times D$，式中，D 为 D_7～D_0 对应的十进制数字。为了产生正弦波，可以建立一个正弦数字量数组，取值范围为一个周期，循环将这些数据送 DAC0832 进行转换，即可在输出端得到正弦波。

源程序

```
/*****************************************************
File name:          DAC0832 数模转换.c
Chip type:          ATmega16
Clock frequency:    8.0MHz
```

**/

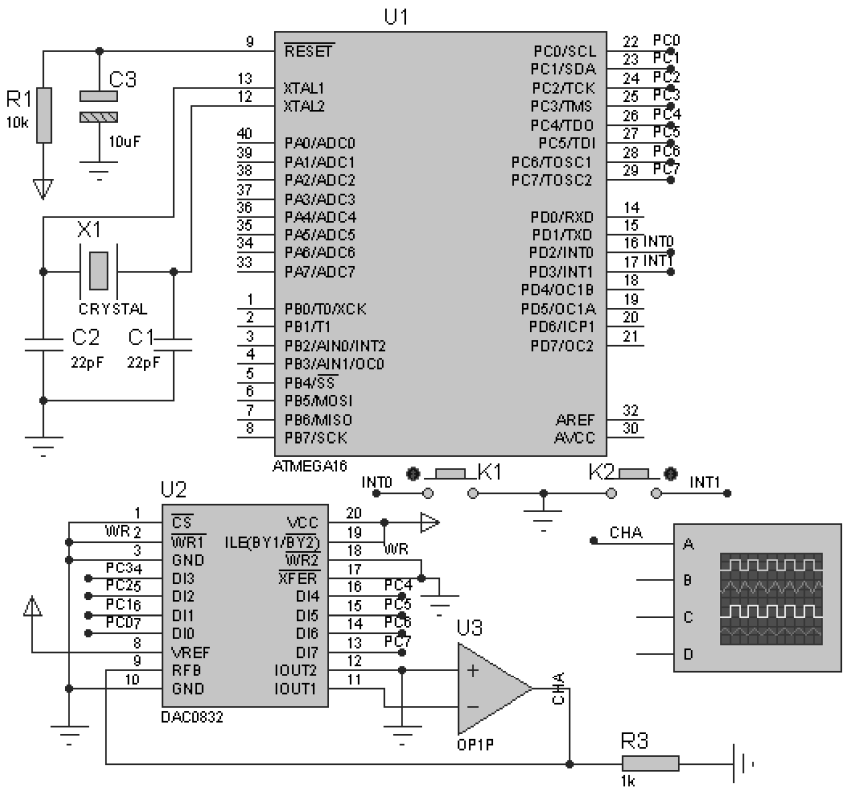


图 14-4　DAC0832 D/A 转换电路图

```
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define   button1        PIND.2
#define   button2        PIND.3
#define   wave           PORTC
uint counter;
uint step;
uchar   tab[256]={                                          //输出正弦波的采样值
        0x80,0x83,0x86,0x89,0x8c,0x8f,0x92,0x95,
        0x98,0x9c,0x9f,0xa2,0xa5,0xa8,0xab,0xae,
        0xb0,0xb3,0xb6,0xb9,0xbc,0xbf,0xc1,0xc4,
        0xc7,0xc9,0xcc,0xce,0xd1,0xd3,0xd5,0xd8,
        0xda,0xdc,0xde,0xe0,0xe2,0xe4,0xe6,0xe8,
        0xea,0xec,0xed,0xef,0xf0,0xf2,0xf3,0xf4,
        0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfc,
        0xfd,0xfe,0xfe,0xff,0xff,0xff,0xff,0xff,
        0xff,0xff,0xff,0xff,0xff,0xff,0xfe,0xfe,
        0xfd,0xfc,0xfc,0xfb,0xfa,0xf9,0xf8,0xf7,
        0xf6,0xf5,0xf3,0xf2,0xf0,0xef,0xed,0xec,
        0xea,0xe8,0xe6,0xe4,0xe3,0xe1,0xde,0xdc,
```

```
        0xda,0xd8,0xd6,0xd3,0xd1,0xce,0xcc,0xc9,
        0xc7,0xc4,0xc1,0xbf,0xbc,0xb9,0xb6,0xb4,
        0xb1,0xae,0xab,0xa8,0xa5,0xa2,0x9f,0x9c,
        0x99,0x96,0x92,0x8f,0x8c,0x89,0x86,0x83,
        0x80,0x7d,0x79,0x76,0x73,0x70,0x6d,0x6a,
        0x67,0x64,0x61,0x5e,0x5b,0x58,0x55,0x52,
        0x4f,0x4c,0x49,0x46,0x43,0x41,0x3e,0x3b,
        0x39,0x36,0x33,0x31,0x2e,0x2c,0x2a,0x27,
        0x25,0x23,0x21,0x1f,0x1d,0x1b,0x19,0x17,
        0x15,0x14,0x12,0x10,0xf,0xd,0xc,0xb,0x9,
        0x8,0x7,0x6,0x5,0x4,0x3,0x3,0x2,0x1,0x1,
        0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
        0x0,0x1,0x1,0x2,0x3,0x3,0x4,0x5,0x6,0x7,
        0x8,0x9,0xa,0xc,0xd,0xe,0x10,0x12,0x13,
        0x15,0x17,0x18,0x1a,0x1c,0x1e,0x20,0x23,
        0x25,0x27,0x29,0x2c,0x2e,0x30,0x33,0x35,
        0x38,0x3b,0x3d,0x40,0x43,0x46,0x48,0x4b,
        0x4e,0x51,0x54,0x57,0x5a,0x5d,0x60,0x63,
        0x66,0x69,0x6c,0x6f,0x73,0x76,0x79,0x7c};
void delay(uint k)
{
   uint m,n;
   for(m=0;m<k;m++)
      {
        for(n=0;n<1140;n++)
          {;}
      }
}
interrupt[EXT_INT0] void int0_isr(void)                    //INT0 中断函数
{
   delay(10);
   if(button1==0)                                          //再次确认按键，软件防抖
      {
        TIMSK=0x00;
        if(step<4096)
           {
             step++;
           }
        else
           {
             step=2;
           }
       TIMSK=0x06;
      }
}
interrupt[EXT_INT1] void int1_isr(void)                    //INT1 中断函数
{
   delay(10);
```

```
   if(button2==0)                          //再次确认按键，软件防抖
      {
        if(step>1)
          {
            step--;
          }
        else
          {
            step=4096;
          }
      }
}
interrupt [TIM1_OVF] void timer0_ovf_isr(void)
 {
    TCNT1H=0xFF;
    TCNT1L=0xFF;
    counter=counter+step;
    wave=tab[(uint)counter>>8];
 }
 void port_init(void)
 {
     DDRC=0xFF;
    PORTC=0x00;
     DDRD=0xF3;
    PORTD=0xFF;
 }
void INT_init(void)                        //中断初始化设置
{
    MCUCR=0x0A;                            //定义 INT0 和 INT1 为下降沿时产生中断
    GICR=0xC0;                             //允许 INT0 和 INT1 产生中断
     #asm("sei")                           //开启总中断
}
void TC1_init(void)                        //T/C1 初始化设置
{
    TCCR1A=0x00;                           //WGM13～WGM10=0000，普通模式；COM1A1、
                                           //COM1A0=00，普通端口操作
                                           //COM1B1、COM1B0=00，普通端口操作
    TCCR1B=0x01;                           //CS12～CS10=100，1 分频
    TCNT1H=0xFF;
    TCNT1L=0xFF;
    TIMSK=0x06;
}
 void main(void)
 {
    port_init();
    INT_init();
    TC1_init();
    step=2;
```

```
        while(1)
          {
            ;
          }
    }
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“DAC0832 数模转换.COFF”文件，然后在 Proteus ISIS 中打开已创建的 DAC0832 数模转换电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试运行状态下，按下按键 K1 或 K2 可改变输出正弦波的频率，虚拟示波器输出的正弦波如图 14-5 所示。

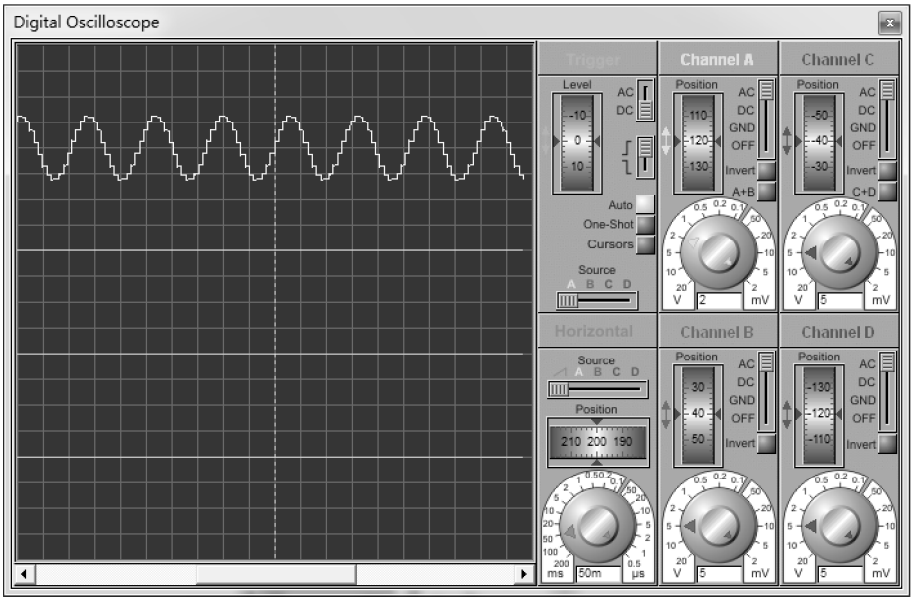


图 14-5　虚拟示波器输出的正弦波

任务 2　TLC5615 D/A 转换

使用单片机和 TLC5615 设计一个波形发生器，要求能够输出锯齿波和三角波。

〖**TLC5615 基础知识**〗

【TLC5615 外形及引脚功能】 TLC5615 的外形封装如图 14-6 所示。其引脚功能如下所述。

☺ DIN：串行数据输入端。

☺ SCLK：串行时钟输入端。

☺ $\overline{CS}$：片选端，低电平有效。

☺ DOUT：用于菊花链的串行数据输出。

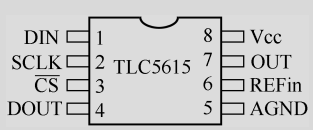


图 14-6　TLC5615 外形封装

☺ AGND：模拟地。

☺ REFin：基准输入端。

☺ OUT：DAC 模拟电压输出端。

☺ Vcc：电源端。

【TLC5615 内部结构及工作原理】 TLC5615 的内部结构如图 14-7 所示。它由 16 位转换寄存器、控制逻辑、10 位 DAC 寄存器、上电复位、DAC、外部基准缓冲器、基准电压倍增器等部分组成。

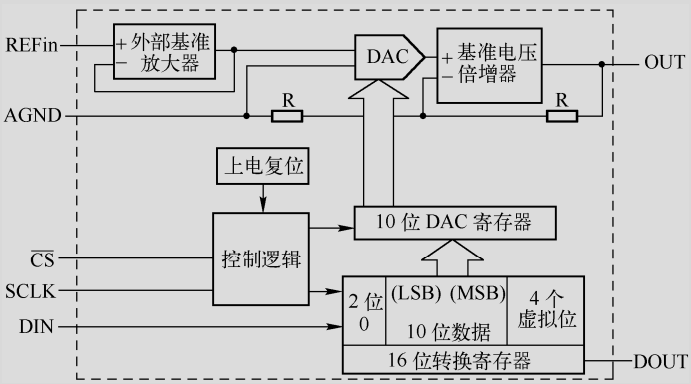


图 14-7　TLC5615 内部结构框图

TLC5615 通过固定增益为 2 的运放缓冲电阻串网络，把 10 位数字数据转换成模拟电压。上电时，内部电路把 DAC 寄存器复位至全 0。其输出具有与基准输入相同的极性，表达式如下：

$$Vo=2\times REFin\times CODE/1024$$

☺ 数据输入：由于 DAC 是 12 位寄存器，所以在写入 10 位数据后，最低 2 位写入 2 个 “0”。

☺ 输出缓冲器：输出缓冲器具有满电源电压幅度（Rail to Rail）输出，它带有短路保护并能驱动有 100pF 负载电容的 2kΩ 负载。

☺ 外部基准：外部基准电压输入经过缓冲，使得 DAC 输入电阻与代码无关。因此，REFin 输入电阻为 10MΩ，输入电容典型值为 5pF，它们与输入代码无关。基准电压决定 DAC 的满度输出。

☺ 逻辑接口：逻辑输入端可使用 TTL 或 CMOS 逻辑电平。但使用满电源电压幅度，CMOS 逻辑可得到最小的功耗。当使用 TTL 逻辑电平时，功耗需求增加约 2 倍。

☺ 串行时钟和更新速率：图 14-8 所示为 TLC5615 的工作时序。TLC5615 的最大串行时钟速率近似为 14MHz。通常，数字更新速率（Digital Update Rate）受片选周期的限制。对于满度输入阶跃跳变，10 位 DAC 建立时间为 12.5μs，这把更新速率限制在 80kHz。

☺ 菊花链接（Daisy-Chaining，即级联）器件：如果时序关系合适，可以在 1 个链路（Chain）中把一个器件的 DOUT 端连接到下一个器件的 DIN 端实现 DAC 的菊花链接（级联）。DIN 端的数据延迟 16 个时钟周期加 1 个时钟宽度后出现在 DOUT 端。DOUT 是低功率的图腾柱（Totem-Poled，即推拉输出电路）输出。当 $\overline{CS}$ 为低电平时，

DOUT 在 SCLK 下降沿变化；当 $\overline{CS}$ 为高电平时，DOUT 保持在最近数据位的值并不进入高阻状态。

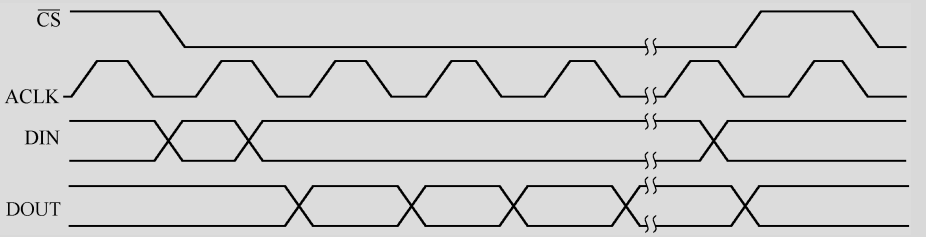


图 14-8　TLC5615 的工作时序

【TLC5615 的使用方法】 当片选信号 $\overline{CS}$ 为低电平时，输入数据读入 16 位移位寄存器（由时钟同步，最高有效位在前）。SCLK 输入的上升沿把数据移入输入寄存器，接着，$\overline{CS}$ 的上升沿把数据传送至 DAC 寄存器。当 $\overline{CS}$ 为高电平时，输入的数据不能由时钟同步送入输入寄存器。所有 $\overline{CS}$ 的跳变应当发生在 SCLK 输入为低电平时。

串行 D/A 转换器 TLC5615 的使用有两种方式，即使用菊花链（级联）功能方式和不使用菊花链（级联）功能方式。

如果不使用菊花链（级联）功能方式时，DIN 只需输入 12 位数据。DIN 输入的 12 位数据中，前 10 位为 TLC5615 输入的 D/A 转换数据，且输入时高位在前，低位在后，后两位必须写入为零的 2 位数值，因为 TLC5615 的 DAC 输入锁存器为 12 位宽。12 位的输入数据序列如下：

D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	0	0

如果使用菊花链（级联）功能时，那么可以传送 4 个高虚拟位（Upper Dummy Bits）在前的 16 位输入数据序列：

4 Upper Dummy	10 Data Bits	0	0

来自 DOUT 的数据需要输入时钟 16 个下降沿，因此需要额外的时钟宽度。当菊花链接（级联）多个 TLC5615 器件时，因为数据传送需要 16 个输入时钟周期加上 1 个额外的输入时钟下降沿数据在 DOUT 端输出，所以，数据需要 4 个高虚拟位。为了提供与 12 位数据转换器传送的硬件与软件兼容性，两个额外位总是需要的。



硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 14-2 所列的元器件，并完成图 14-9 所示的硬件电路图设计。

表 14-2　TLC5615 D/A 转换所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	晶振 CRYSTAL 8MHz	电解电容 CAP-ELEC
电阻 RES	数模转换 TLC5615	运放 LM358	按钮 BUTTON
可调电阻 POT-HG			

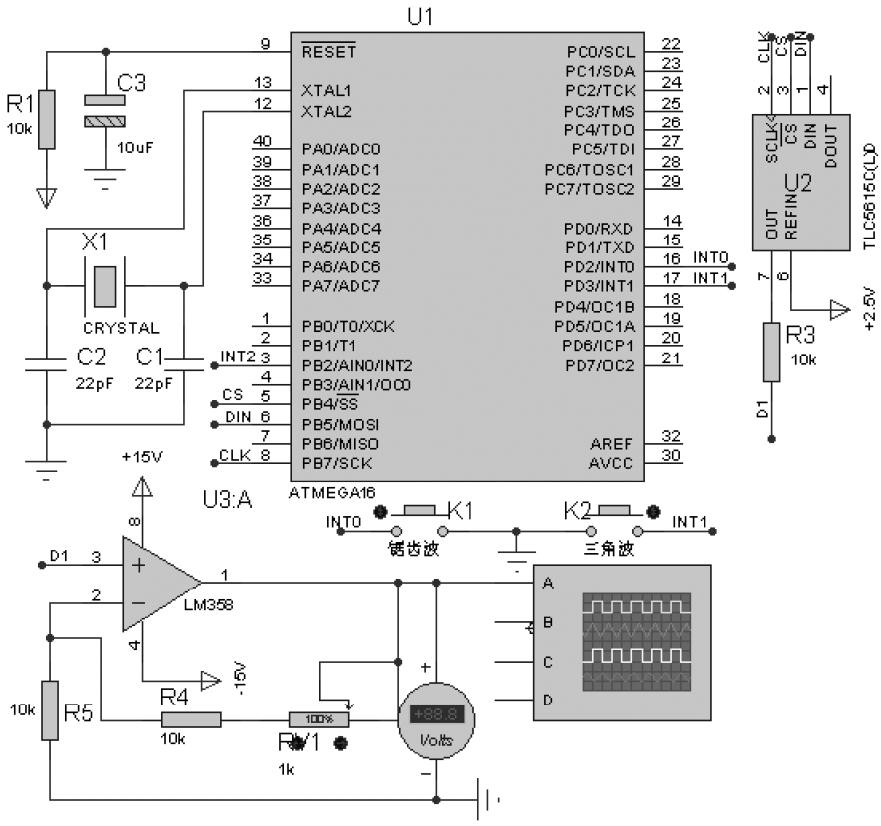


图 14-9　TLC5615 D/A 转换电路图

软件设计

本任务使用按键 K1 和 K2 作为 INT0 和 INT1 的外部中断源，即按键 K1 选择锯齿波、按键 K2 选择三角波。波形的输出都是通过 TLC5615 进行 D/A 转换实现的，由于 TLC5615 为 10 位 D/A 转换器，要输出锯齿波，首先判断 i 是否为 1024（即 2^{10}），若是则将 i 值通过 TLC5615 转换，并将 i 加 1，否则将 i 清零，再判断 i 值。要输出三角波，首先判断 i 是否小于 1000，若是，则将 i 值通过 TLC5615 转换，并将 i 加 1，否则再判断 i 是否 1000，若是则 i 减 1 并判断是否大于 1，且将 i 值通过 TLC5615 转换即可。使用 SPI 串行总线输出数据时，首先需输出高 2 位，然后再输出低 8 位。

源程序

```
/*****************************************************
  文件名:          TLC5615 数模转换.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define   button1          PIND.2
```

```
#define   button2          PIND.3
#define   CS               PORTB.4
uint  dat1;          //将要输入的数据   大小
uchar     a;
void delay(uint k)
{
   uint m,n;
   for(m=0;m<k;m++)
      {
         for(n=0;n<1140;n++)
           {;}
      }
}
interrupt[EXT_INT0] void int0_isr(void)                    //INT0 中断函数
{
   delay(10);
   if(button1==0)                                          //再次确认按键，软件防抖
      {
       a=0;
      }
}
interrupt[EXT_INT1] void int1_isr(void)                    //INT1 中断函数
{
   delay(10);
   if(button2==0)                                          //再次确认按键，软件防抖
      {
         a=1;
      }
}
void SPI_MasterTransmit(uchar i)                           //SPI 数据发送
{
   SPDR = i;                                               //启动数据串行输出
   while(!(SPSR==0x80))                                    //SPIF 等待传输结束
      {
        ;
      }
}
void SPI_MasterInit(void)          //SPI 总线发送初始化
 {
    SPCR=0x53;                     //SPE=1，MSTR=1，SPR1=1，SPR0=1，使能 SPI 主机模式，
                                   //设置时钟速率为 fck/128
 }
void wave_out(void)
{
   uint dat;
   dat=dat1;
   dat<<=2;                        //为适应 TLC5615 格式，dat 要先左移 2 位
   CS=0;                           //CS 使能 SPI
```

```
    SPI_MasterTransmit(dat>>8);          //发送高 8 位
    SPI_MasterTransmit(dat);             //发送低 8 位
    CS=1;                                //CS 禁用 SPI
}
void corr1(void)
{
    uint i;
    for(i=0;i<1024;i++)
          {
            dat1=i;
            wave_out();
          }
}
void corr2(void)                         //三角波
{
    uint i;
    for(i=0;i<1000;i++)
          {
            dat1=i;
            wave_out();
          }
    for(i=1000;i>0;i--)
          {
            dat1=i;
            wave_out();
          }
}
void port_init(void)                     //端口初始化
{
      DDRB=0xFB;
     PORTB=0xF0;
      DDRD=0xF3;
     PORTD=0xFF;
}
void INT_init(void)                      //中断初始化设置
{
     MCUCR=0x0A;                         //定义 INT0 和 INT1 为下降沿时产生中断
      GICR=0xC0;                         //允许 INT0、INT1 中断
      #asm("sei")                        //开启总中断
}
void main(void)
{
    port_init();
    INT_init();
    SPI_MasterInit();
    while(1)
       {
          switch(a)
```

```
        {
          case 0:   corr1();break;
          case 1:   corr2();break;
        }
        }
}
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“TLC5615 数模转换.COFF”文件，然后在 Proteus ISIS 中打开已创建的 TLC5615 数模转换电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试运行状态下虚拟示波器输出的波形的如图 14-10 所示。

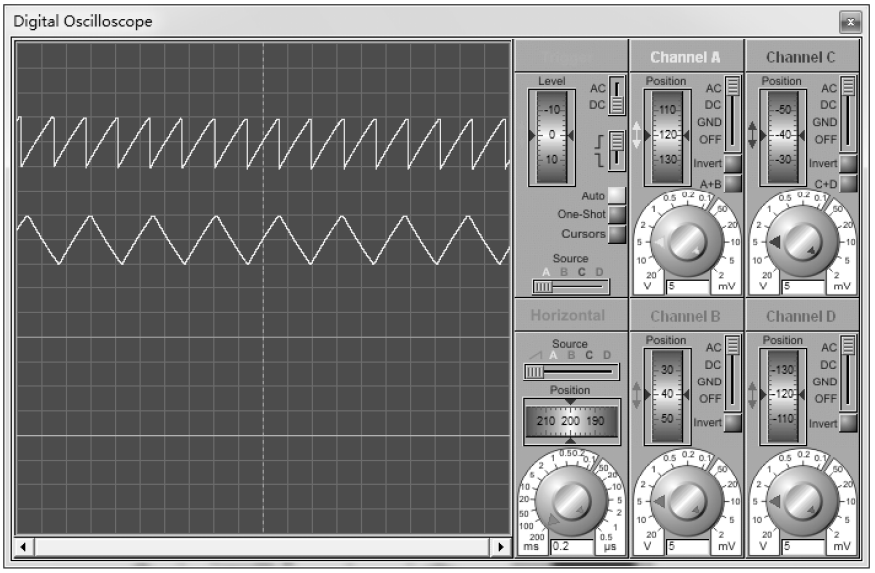


图 14-10　虚拟示波器输出的波形

任务 3　MAX515 D/A 转换

通过调节可调电阻 RV1 向 ATmega16 的 D/A 转换器输入模拟电压，单片机将 A/D 转换后的数字量输出给 MAX515，经 MAX515 进行 D/A 转换后控制 LED 的亮度。

硬件设计

MAX515 是由美信（Maxim）公司生产的一种低功耗、电压输出型 10 位串行 D/A 转换器，它与 TLC5615 完全兼容，可以相互替代。

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 14-3 所列的元器件，并完成图 14-11 所示的硬件电路图设计。

表 14-3　MAX515 D/A 转换所用元器件

单片机 ATmega16	瓷片电容 CAP 22pF	晶振 CRYSTAL 8MHz	电解电容 CAP-ELEC
电阻 RES	数模转换 MAX515	发光二极管 LED-GREEN	按钮 BUTTON
可调电阻 POT-HG			

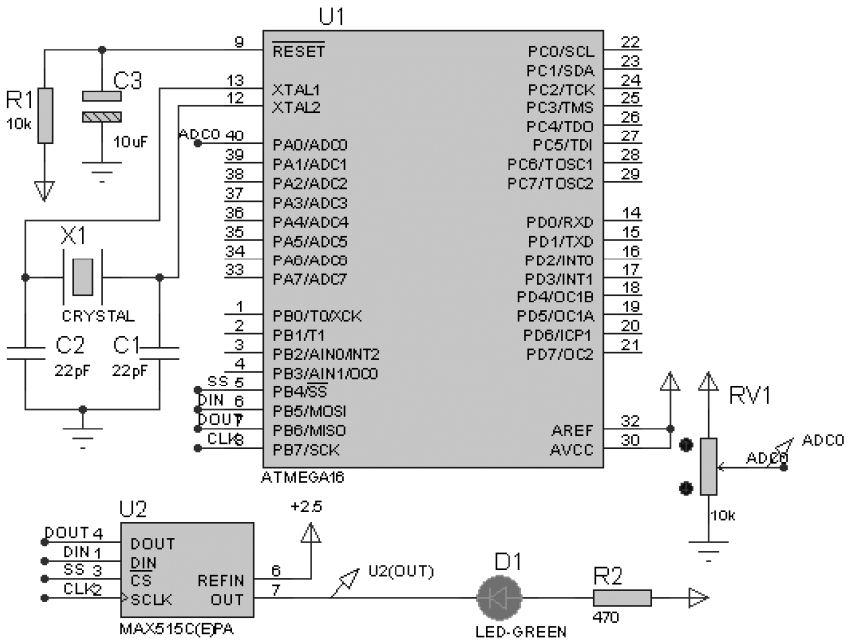


图 14-11　MAX515 D/A 转换电路图

软件设计

单片机首先通过内置 A/D 转换器将 RV1 输入的模拟电压转换为相应为的数值，然后该数值通过 SPI 总线接口传输到 MAX515 即可。

源程序

```
/*****************************************************
  文件名:          MAX515 数模转换.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
#define ulong unsigned long
#define   CS          PORTB.4
uchar cnt;
uint adc_data;
interrupt [TIM0_COMP] void timer0_comp_isr(void)      //T/C0 比较匹配中断服务
{
```

```
    cnt=0x01;
}
interrupt [ADC_INT] void adc_isr(void)        //ADC 转换完成中断服务
{

    adc_data=ADCW;                            //读取 ADC 转换结果
}
void SPI_MasterTransmit(uchar i)              //SPI 数据发送
{
  SPDR = i;                                   //启动数据串行输出
  while(!(SPSR= =0x80))                       //SPIF 等待传输结束
     {
       ;
     }
}
void SPI_MasterInit(void)                     //SPI 总线发送初始化
 {
    SPCR=0x53;                                //SPE=1，MSTR=1，SPR1=1，SPR0=1，使能 SPI 主
                                              //机模式，设置时钟速率为 fck/128
 }
void wave_out(void)
{
  uint dat;
  dat=adc_data;
  dat<<=2;                                    //为适应 TLC5615 格式，dat 要先左移 4 位
  CS=0;                                       //CS 使能 SPI
  SPI_MasterTransmit(dat>>8);                 //发送高 8 位
  SPI_MasterTransmit(dat);                    //发送低 8 位
  CS=1;                                       //CS 禁用 SPI
}
 void port_init(void)
 {
    DDRA=0x00;                                //模拟量由 PA0 输入
    PORTA=0x00;
    DDRB=0xFF;                                //LED 片选控制
    PORTB=0x00;
 }
void timer0_init(void)
 {
    TCCR0=0x0B;                               //内部时钟，64 分频（8M/64=125kHz），CTC 模式
    TCNT0=0x00;
    OCR0=0xF9;                                //OCR0 = 0xF9（即 249），(249+1)/125=2ms
    TIMSK=0x02;                               //允许 T/C0 比较中断
 }
 void ADC_init(void)
 {
    ADMUX=0x40;                               //参考电源 AVcc、ADC0 单端输入
    SFIOR&=0x1F;
```

```
    SFIOR|=0x60;                    //选择 T/C0 比较匹配中断为 ADC 触发源
    ADCSRA=0xAE;                    //ADC 允许、自动触发转换、ADC 转换中断允许、
                                    //ADCclk=8000000Hz/64=125kHz
    #asm("sei")                     //开放全局中断
}
void main(void)
{
    port_init();
    timer0_init();
    ADC_init();
    SPI_MasterInit();
    while(1)
      {
        if(cnt==1)
          {
            wave_out();
            cnt=0;
          }
      }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“MAX515 数模转换.COFF”文件，然后在 Proteus ISIS 中打开已创建的 MAX515 数模转换电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试状态。在调试运行状态下，滑动 RV1 可调电阻，可改变 LED 的亮度，其仿真效果如图 14-12 所示。

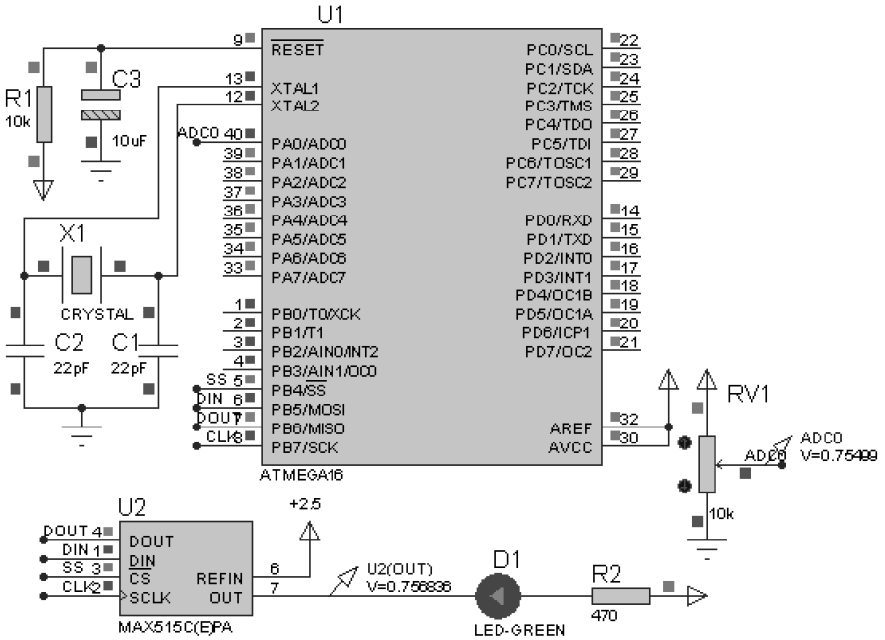


图 14-12　MAX515 D/A 转换仿真效果图

项目十五　综合应用设计

〖知识目标〗
☺ 熟悉单片机定时器的编程方法;
☺ 灵活运用单片机 C 语言中的字符串、数组、指针;
☺ 掌握矩阵键盘的 C 语言编程。
〖能力目标〗
☺ 结合硬件电路，进行系统软件设计的分析;
☺ 编写程序，进行硬件、软件综合调试，实现设计的全部功能要求。

任务 1　篮球计分器的设计

设计一个篮球计分器，能够显示比赛时间、甲队的乙队的得分。甲队和乙队的得分分别有加 1 分、加 2 分、加 3 分、减 1 分、比分清零、比分切换操作；比赛时间采用 10min 倒计时，可以进行加时或减时 1s 操作，还可以暂停计时及比赛时间复位等操作。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 15-1 所列的元器件，并完成图 15-1 所示的硬件电路图设计。

表 15-1　篮球计分器所用元器件

单片机 ATmega16	电解电容 CAP-ELEC 10uF	瓷片电容 CAP22pF	电阻 RES
晶振 CRYSTAL 8MHz	数码管 7SEG-MPX2-CA-BLUE	三极管 NPN	按钮 BUTTON

软件设计

篮球计分器有时间显示、甲队得分显示及乙队得分显示，因此可以定义 8 个显示暂存数组 dis_buff[8]，其中 dis_buff[0]和 dis_buff[1]用于甲队得分显示；dis_buff[2]和 dis_buff[3]用于乙队得分显示；dis_buff[4]～dis_buff[1]用于时间显示。由于篮球计分器的控制按键较多，采用 4×4 矩阵键盘可以实现这些功能操作，所以应有键盘扫描函数 keyscan()。在键盘扫描函数中完成键盘扫描操作时，获取按键值 key。然后在键值处理函数 key_op()中根据 key 值完成计分、计时等操作。而 data_in()函数将显示数值进行整定。

源程序

```
/*************************************************
```

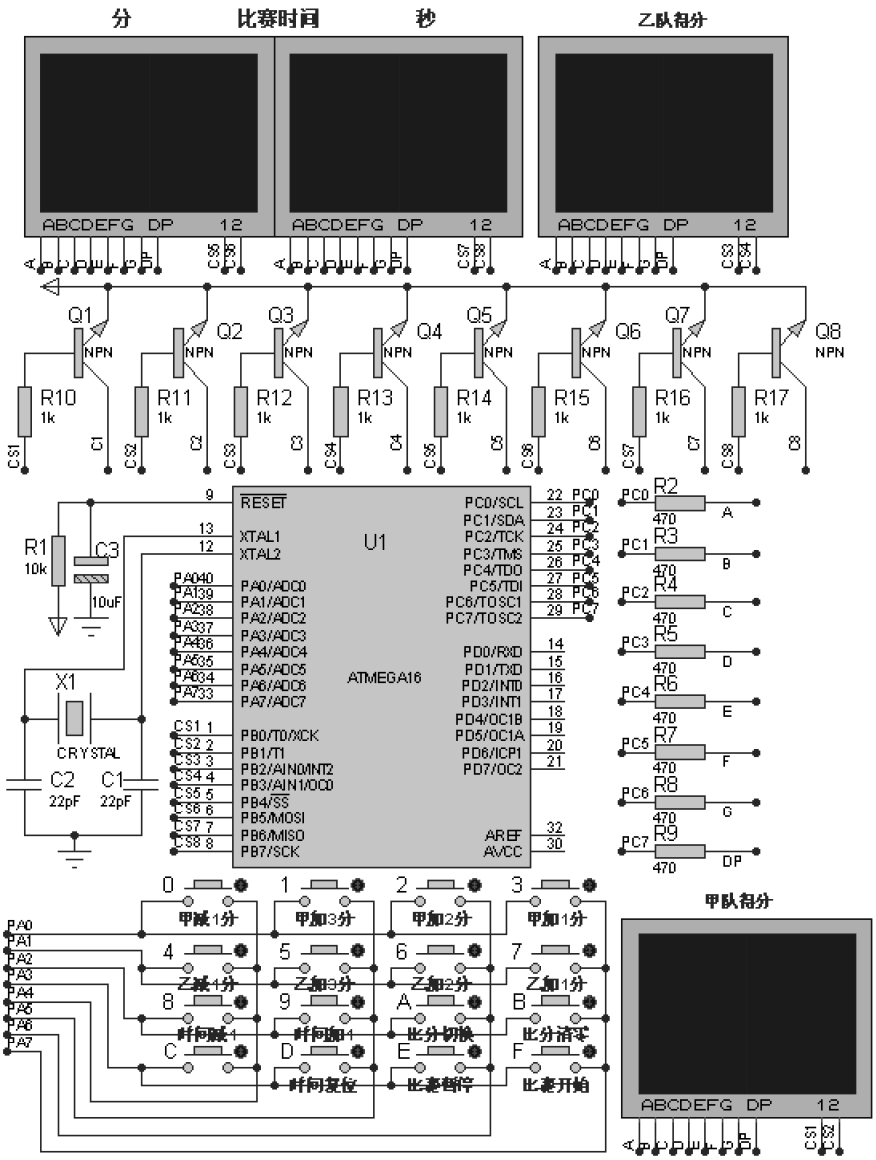


图 15-1　篮球计分器电路图

```
 File name:          篮球计分器.c
 Chip type:          ATmega16
 Clock frequency:    8.0MHz
*************************************************/
#include "mega16.h"
#define uchar unsigned char
#define uint unsigned int
flash uchar tab[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,      //共阳极 LED  0～F 的段码
                   0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E,0xBF,};
uchar dis_buff[8];                  //定义暂存显示数据
bit key_flag;                       //定义按键状态
uint a,a_H,a_L;                     //甲队比分
```

```
uint b,b_H,b_L;                          //乙队比分
uchar min=10,min_H,min_L;                //分
uchar sec=0,sec_H,sec_L;                 //秒
uchar cnt;
void    delay(uint k)                    //延时约 0.1ms
{
    uint    m,n;
    for(m=0;m<k;m++)
     {
        for(n=0;n<114;n++);
     }
}
interrupt [TIM2_OVF] void timer2_ovf_isr(void)
{
    cnt++;
    TCNT2=0x62;
}
void    times_op(void)                   //1s 时间整定
{
  if(cnt==200)
     {
        cnt=0;
        if(sec==0)
           {
              sec=59;
              min--;
           }
        else
           {
              sec--;
           }
     }
}
uchar ScanKey()                          //键盘扫描
{
  uchar temp,temp1,key;
  temp=PINA;                             //读取 PA 端口状态
  temp&=0x0F;                            //判断行线状态
  switch(temp)
  {
    case 0x0E:                           //行线的 PA0 为低电平
      {
           DDRA=0x0F;                    //改变端口方向，准备进行列线的状态判断
          PORTA=0xF0;
           delay(5);                     //延时片刻，以利于状态的获取
          temp1=PINA;                    //读取 PA 端口状态
         temp1&=PINA;                    //判断列线状态
         switch(temp1)
```

```
        {
         case 0x70:                    //列线的 PA7 为低电平，表示“0”键按下
             key=0;                    //键值为 0
             break;                    //跳出列线状态判断
         case 0xB0:                    //列线的 PA6 为低电平，表示“1”键按下
             key=1;                    //键值为 1
             break;                    //跳出列线状态判断
         case 0xD0:                    //列线的 PA5 为低电平，表示“2”键按下
             key=2;
             break;
         case 0xE0:                    //列线的 PA4 为低电平，表示“3”键按下
             key=3;
             break;
         default:                      //列线没发生状态改变
             break;
        }
       DDRA=0xF0;                      //改变 PA 端口方向，为下次行线判断做好准备
       PORTA=0x0F;
       break;
   }
case 0x0D:                             //行线的 PA1 为低电平
   {
        DDRA=0x0F;                     //改变端口方向，准备进行列线的状态判断
       PORTA=0xF0;
        delay(5);
       temp1=PINA;                     //读取 PD 端口状态
      temp1&=PINA;                     //判断列线状态
      switch(temp1)
        {
         case 0x70:                    //列线的 PA7 为低电平，表示“4”键按下
           key=4;
           break;
         case 0xB0:                    //列线的 PA6 为低电平，表示“5”键按下
           key=5;
           break;
         case 0xD0:                    //列线的 PA5 为低电平，表示“6”键按下
           key=6;
           break;
         case 0xE0:                    //列线的 PA4 为低电平，表示“7”键按下
           key=7;
           break;
        default:                       //列线没发生状态改变
           break;
        }
       DDRA=0xF0;                      //改变 PA 端口方向，为下次行线判断做好准备
       PORTA=0x0F;
       break;
   }
```

```
case 0x0B:                          //行线的 PA2 为低电平
  {
      DDRA=0x0F;                    //改变端口方向，准备进行列线的状态判断
     PORTA=0xF0;
      delay(5);
     temp1=PINA;                    //读取 PD 端口状态
    temp1&=PINA;                    //判断列线状态
    switch(temp1)
      {
       case 0x70:                   //列线的 PA7 为低电平，表示“8”键按下
         key=8;
         break;
       case 0xB0:                   //列线的 PA6 为低电平，表示“9”键按下
         key=9;
         break;
       case 0xD0:                   //列线的 PA5 为低电平，表示“A”键按下
         key=10;
         break;
       case 0xE0:                   //列线的 PA4 为低电平，表示“B”键按下
         key=11;
         break;
      default:                      //列线没发生状态改变
         break;
      }
     DDRA=0xF0;                     //改变 PA 端口方向，为下次行线判断做好准备
     PORTA=0x0F;
     break;
  }
case 0x07:                          //行线的 PA3 为低电平
  {
      DDRA=0x0F;                    //改变端口方向，准备进行列线的状态判断
     PORTA=0xF0;
      delay(5);
     temp1=PINA;                    //读取 PA 端口状态
    temp1&=PINA;                    //判断列线状态
    switch(temp1)
      {
       case 0x70:                   //列线的 PA7 为低电平，表示“C”键按下
         key=12;
         break;
       case 0xB0:                   //列线的 PA6 为低电平，表示“D”键按下
         key=13;
         break;
       case 0xD0:                   //列线的 PA5 为低电平，表示“E”键按下
         key=14;
         break;
       case 0xE0:                   //列线的 PA4 为低电平，表示“F”键按下
         key=15;
```

```
                break;
               default:                          //列线没发生状态改变
                break;
              }
              DDRA=0xF0;                         //改变 PA 端口方向，为下次行线判断做好准备
              PORTA=0x0F;
              break;
          }
          default:                               //行线没发生状态改变
          break;
    }
    return(key);                                 //返回键值
}
void key_op(void)                                //键值处理
{
    uchar   temp,keynum;
    uchar   e,f;                                 //定义比分切换变量
    temp=PINA;                                   //判断是否有按键
    temp&=0x0F;
    if(temp==0x0F)
      {
        key_flag=0;
      }
    else
      {
        delay(100);                              //约 10ms 延时去抖
        temp=PINA;                               //判断是否有按键
        temp&=0x0F;}
    if(temp==0x0F)
      {
        key_flag=0;
      }
    else
      {
        key_flag=1;
        keynum=ScanKey();                        //键值送 keynum
      }
    while(temp!=0x0F)
      {
        temp=PINA;
        temp&=0x0F;
      }
    if(key_flag==1)
      {
        switch(keynum)
          {
            case 0:                              //甲队比分加 1
              {
```

```
    a++;
    if(a>=100)
      {
        a=99;
      }
    break;
  }
case 1:                    //甲队比分加 2
  {
    a=a+2;
    if(a>=100)
      {
        a=99;
      }
    break;
  }
case 2:                    //甲队比分加 3
  {
    a=a+3;
    if(a>=100)
      {
        a=99;
      }
    break;
  }
case 3:                    //甲队比分减 1
  {
    a--;
    if(a<=0)
      {
        a=0;
      }
    break;
  }
case 4:                    //乙队比分加 1
  {
    b++;
    if(b>=100)
      {
        b=99;
      }
    break;
  }
case 5:                    //乙队比分加 2
  {
    b=b+2;
    if(b>=100)
      {
```

```
            b=99;
          }
      break;
   }
case 6:                          //乙队比分加 3
   {
      b=b+3;;
      if(b>=100)
          {
            b=99;
          }
      break;
   }
case 7:                          //乙队比分减 1
   {
      b--;;
      if(b<=0)
          {
            b=0;
          }
      break;
   }
case 8:                          //比分清零
   {
      a=0;
      b=0;
      break;
   }
case 9:                          //双方比分切换
   {
      e=a;
      f=b;
      a=f;
      b=e;
      break;
   }
case 10:                         //定时时间加 1
   {
      min++;
        {
          if(min==99)
          {
             min=0;
          }
        }
      break;
   }
case 11:                         //定时时间减 1
```

```
                {
                  min--;;
                  if(min<=0)
                    {
                      min=99;
                    }
                  break;
                }
               case 12:                          //比赛开始
                {
                  TIMSK=0x40;                    //启动 T/C2 计时
                  break;
                }
               case 13:                          //比赛暂停
                {
                  TIMSK=0x00;                    //暂停 T/C2 计时
                  break;
                }
               case 14:                          //时间复位
                {
                  min=10;
                  sec=0;
                  TIMSK=0x00;                    //暂停 T/C2 计时
                  break;
                }

            }
        }
}
void   data_in(void)                             //显示数值整定
{
  key_op();
  times_op();
  a_H=a/10;
  a_L=a%10;
  b_H=b/10;
  b_L=b%10;
  min_H=min/10;
  min_L=min%10;
  sec_H=sec/10;
  sec_L=sec%10;
  dis_buff[0]=a_H;
  dis_buff[1]=a_L;
  dis_buff[2]=b_H;
  dis_buff[3]=b_L;
  dis_buff[4]=min_H;
  dis_buff[5]=min_L;
  dis_buff[6]=sec_H;
```

```
    dis_buff[7]=sec_L;
}
void display()
{
    uchar i,j;
    data_in();
    j=0x01;
    for(i=0;i<8;i++)
        {
          PORTB=0x00;
          PORTC=tab[dis_buff[i]];
          PORTB=j;
          j=j<<1;
          delay(1);
        }
}
void timer2_init(void)
{
    TCCR2=0x04;                    //WGM01=0 WGM00=0 选择普通模式;
                                   //COM01=0 COM00=0 使 PB3 为通用 I/O（OC0 不与
                                   //引脚相连）CS02=1 CS01=0 CS01=0 选择 256 分频
     TCNT2=0x62;                   //98   t=(255-TCNT2)*256/8000000   t=0.005024,TCNT2=98
     #asm("sei");                  //开启总中断
}
void port_init(void)               //端口初始化
{
    PORTA=0x0F;                    //PA 端口初始值设置
    DDRA=0xF0;                     //PA 端口配置
    PORTB=0xFE;                    //PB 端口初始值设置
    DDRB=0xFF;                     //PB 端口配置为输出
    PORTC=0xFF;                    //PC 端口初始值设置
    DDRC=0xFF;                     //PC 端口配置为输出
}
void main(void)
{
    port_init();
    timer2_init();
    while (1)
        {
          display();
        }
}
```



调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“篮球计分器.COFF”文件，然后在 Proteus ISIS 中打开已创建的篮球计分器电路图并进行相应设置。单击 Proteus ISIS 模拟调试

按钮中的运行按钮，进入调试状态。刚运行时，LED 数码管显示的时间为“1000”，表示比赛的分节时间为 10min；LED 数码管显示的甲队和乙队得分均为 0。当按下比赛开始按键后，比赛时间进行倒计时，操作按键相应的得分按钮，来控制甲队或乙队的得分情况，其仿真效果如图 15-2 所示。

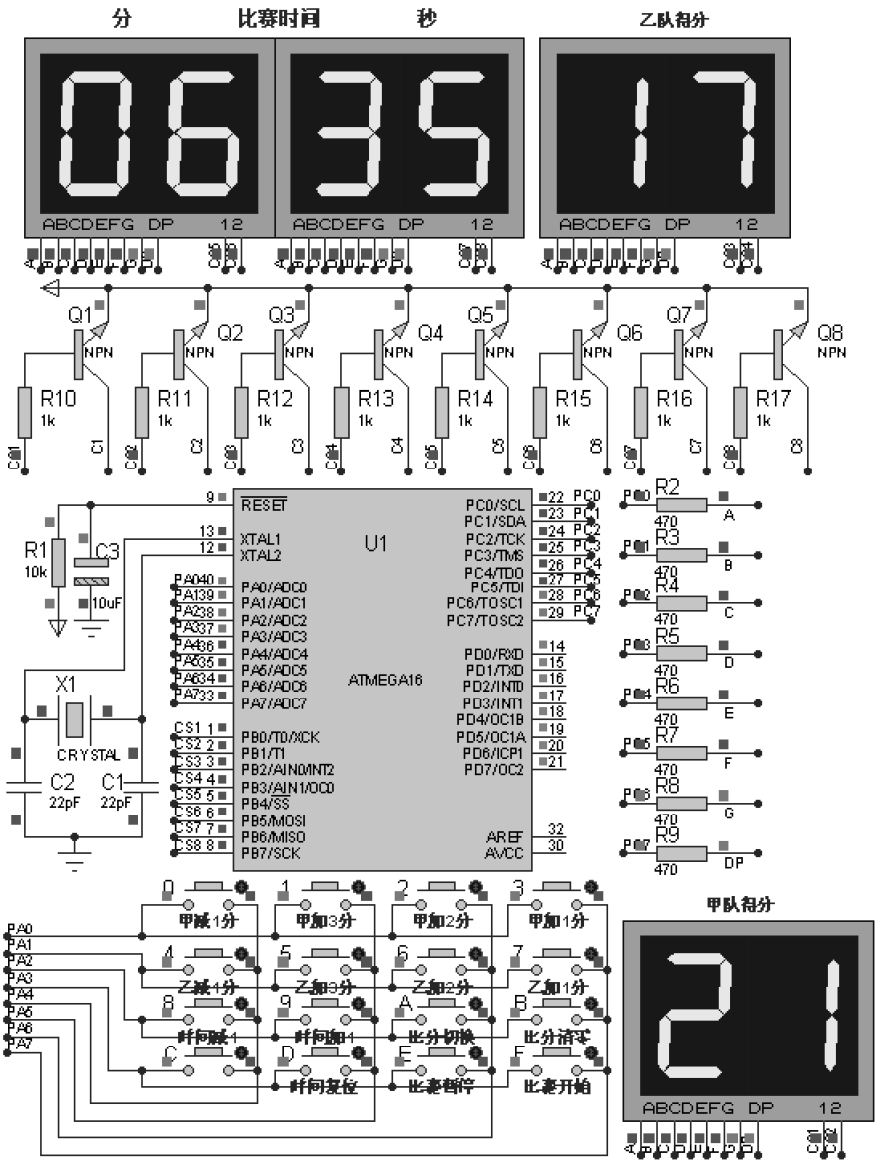


图 15-2　篮球计分器仿真效果图

任务 2　按键选播电子音乐

按键 K 与单片机的 INT0 连接，要求每按一次按键 K，单片机控制蜂鸣器发出不同的电子音乐，同时数码管显示电子音乐的序号。按完 4 次后，再重复循环。这 5 首歌电子音乐分

别为《送别》、《两只老虎》、《哈巴狗》、《兰花草》、《不倒翁》。

〖电子音乐的基础知识〗

【音频脉冲的产生】 音乐的产生主要是通过单片机的 I/O 端口输出高低不同的脉冲信号来控制蜂鸣器发音。要产生音频脉冲信号，只要算出某一音频的周期（1/频率），然后将此周期除以 2，即为半周期的时间。利用单片机定时器计时这个半个周期时间，每当计时到后就将输出脉冲的 I/O 口反相，然后重复计时此半周期时间再对 I/O 端口反相，这样就能在此 I/O 端口上得到此频率的脉冲。

通常利用 ATmega16 单片机的内部定时器 1，工作在普通模式下，改变计数初值 TCNT1H 和 TCNT1L 来产生不同的频率。

例如，若单片机采用 8MHz 的晶振，不使用预分频，要产生频率为 587Hz 的音频脉冲时，其音频脉冲信号的周期 t=1/587=1703.5775μs≈1704μs，半周期的时间为 852μs，因此只要令计数器每计数 6814 次时将 I/O 端口反相，就可得到 C 调中音 Re。

在没使用预分频时，T1 的计数脉冲初值与频率的关系为

$$T=65535-((1/f_r)\div 2\times f_{osc})$$

式中，T 为计数初值；f_r 为要产生的频率；f_{osc} 为系统时钟频率

例如，设 f_{osc}=为 8MHz，求低音 Do（262Hz），中音 Do（523Hz）和高音 Do（1046Hz）的计数值。

解：根据 $T=65535-((1/F_r)\div 2\times f_{osc})=65536-((1/f_r)\div 2\times 8000000)$

低音 Do 的 T=65536−15267=50268

中音 Do 的 T=65536−7648=57887

高音 Do 的 T=65536−3824=61711

综上所述，在 8MHz 频率下，C 调各音符频率计数值 T 的关系见表 15-2。

表 15-2 C 调各音符频率与计数值 *T* 的关系

音 符	频率/Hz	简谱码（*T* 值）	音 符	频率/Hz	简谱码（*T* 值）
低 1 Do	262	50268	中 5 So	784	60433
低 2 Re	294	51931	中 6 La	880	60990
低 3 Mi	330	53414	中 7 Si	988	61487
低 4 Fa	349	54074	高 1 Do	1046	61711
低 5 So	392	55331	高 2 Re	1175	62119
低 6 La	440	56444	高 3 Mi	1318	62500
低 7 Si	494	57438	高 4 Fa	1397	62672
中 1 Do	523	57887	高 5 So	1568	62984
中 2 Re	587	58721	高 6 La	1760	63262
中 3 Mi	659	59465	高 7 Si	1967	63502
中 4 Fa	698	59804			

【音乐节拍的产生】 每个音符使用 1 字节，字节的高 4 位代表音符的高低，低 4 位代表音符的节拍，表 15-3 为节拍与节拍码的对照表。如果 1 拍为 0.4s，1/4 拍是 0.1s，只要设定延迟时间就可求得节拍的时间。假设 1/4 拍为 1DELAY，那么 1 拍应为 4DELAY，

以此类推。所以只要求得 1/4 拍的 DELAY 时间，其余的节拍就是它的倍数。表 15-4 所列为 1/4 和 1/8 节拍的时间设定。

表 15-3　节拍与节拍码对照

节拍码	节拍数	节拍码	节拍数
1	1/4 拍	1	1/8 拍
2	2/4 拍	2	1/4 拍
3	3/4 拍	3	3/8 拍
4	1 拍	4	1/2 拍
5	$1\frac{1}{4}$ 拍	5	5/8 拍
6	$1\frac{1}{2}$ 拍	6	3/4 拍
8	2 拍	8	1 拍
A	$2\frac{1}{2}$ 拍	A	$1\frac{1}{4}$ 拍
C	3 拍	C	$1\frac{1}{2}$ 拍
F	$3\frac{3}{4}$ 拍		

表 15-4　1/4 和 1/8 节拍的时间设定

1/4 节拍的时间设定		1/8 节拍的时间设定	
曲调值	DELAY	曲调值	DELAY
调 4/4	125ms	调 4/4	62ms
调 3/4	187ms	调 3/4	94ms
调 2/4	250ms	调 2/4	125ms

【移调】 一般的歌曲，有 3/8、2/4、3/4、4/4 等节拍类型，但不管有几拍，基本上是在 C 调下演奏的。如果是 C 调，则音名 C 唱 Do，音名 D 唱 Re，音名 E 唱 Mi，音名 F 唱 Fa，音名 G 唱 So，音名 A 唱 La，音名 B 唱 Ti 等。但并不是所有的歌曲都是在 C 调下演奏的，还有 D 调、E 调、F 调、G 调等。D 调是将 C 调各音符上升一个频率实现的，即 C 调下的音名 D 在 D 调下唱 Do，C 调下的音名 E 在 D 调下唱 Re，C 大调的音名 F 在 D 调下升高半音符 F#唱 Mi，C 调下的音名 G 在 D 调下唱 Fa，C 调下的音名 A 在 D 调下唱 So，C 调下的音名 B 在 D 调下唱 La，C 调下的音名 C 在 D 调下升高半音 C#符唱 Ti。这种改变唱法称为移调。

E 调是在 D 调的基础上进行移调的，而 F 调是在 E 调的基础上进行移调的……表 15-5 所列为各调音符与音名的关系。

表 15-5　各大调的音符与音名的关系

音名 / 调	Do	Re	Mi	Fa	So	La	Ti
C 调	C	D	E	F	G	A	B
D 调	D	E	F#	G	A	B	C
E 调	E	F#	G#	A	B	C	D
F 调	F	G	A	B	C	D	E

续表

音名 / 调	Do	Re	Mi	Fa	So	La	Ti
G 调	G	A	B	C	D	E	F#
A 调	A	B	C#	D	E	F#	G#
B 调	B	C	D	E	F	G	A

【音乐软件的设计】

☺ 音乐代码库的建立方法

▻ 先找出乐曲的最低音和最高音范围，然后确定音符表 T 的顺序。

▻ 把 T 值建立在 TABLE1，构成发音符的计数值放在“TABLE1”

▻ 简谱码（音符）为高 4 位，节拍（节拍数）为低 4 位，音符节拍码放在程序的“TABLE”处。

▻ 音符节拍码 0xFF 为音乐结束标志。

☺ 选曲：在一个程序中，若需演奏两首或两首以上的歌曲时，音乐代码库的建立有两种方法：一种方法是将每首歌曲建立相互独立的音符表 T 和发音符计数值 TABLE。另一种方法是在建立共用的音符表 T 后，再写每首歌的发音计数值 TABLE 中的代码。不管采用哪种方法，每首歌曲结束时，在 TABLE 中均需加上音乐结束符 0xFF。

硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 15-6 所列的元器件，并完成图 15-3 所示的硬件电路图设计。

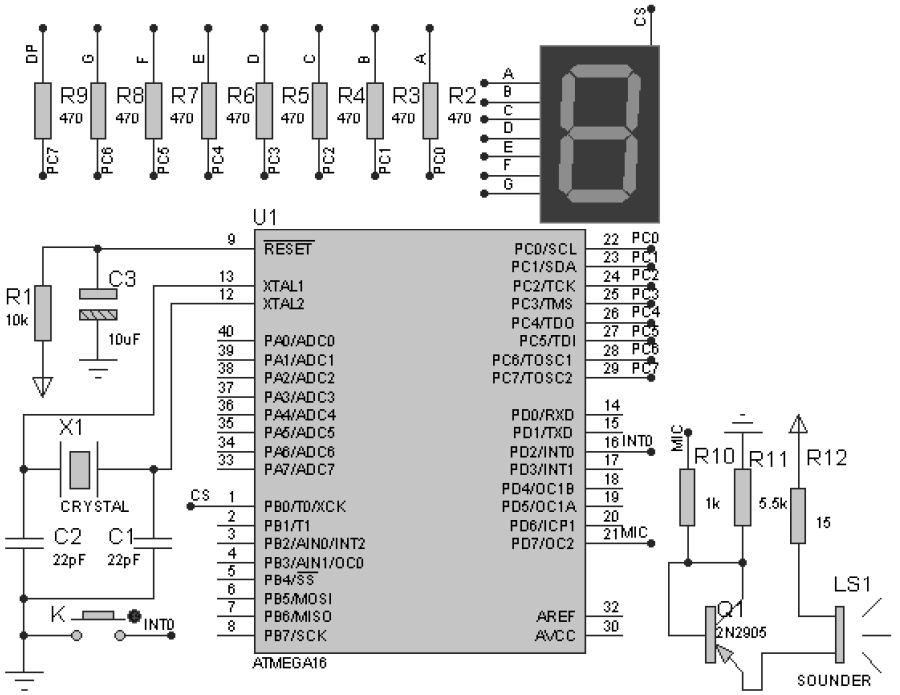


图 15-3　按键选播电子音乐电路图

表 15-6　按键选播电子音乐所用元器件

单片机 ATmega16	电解电容 CAP-ELEC 10uF	瓷片电容 CAP22pF	电阻 RES
晶振 CRYSTAL 8MHz	数码管 7SEG-MPX-CA-BLUE	三极管 2N2905	按钮 BUTTON
蜂鸣器 SOUNDER			

软件设计

这 5 首电子音乐的歌曲如图 15-4 所示，从图中可以看出这 5 首歌曲的使用的音符较多，所以可以将表 15-2 中的所有 T 值都放在 Tone_tab[]中，然后将每首歌的音符放在相应的 song_Tone 中，并将节拍数放在 song_Time 中。

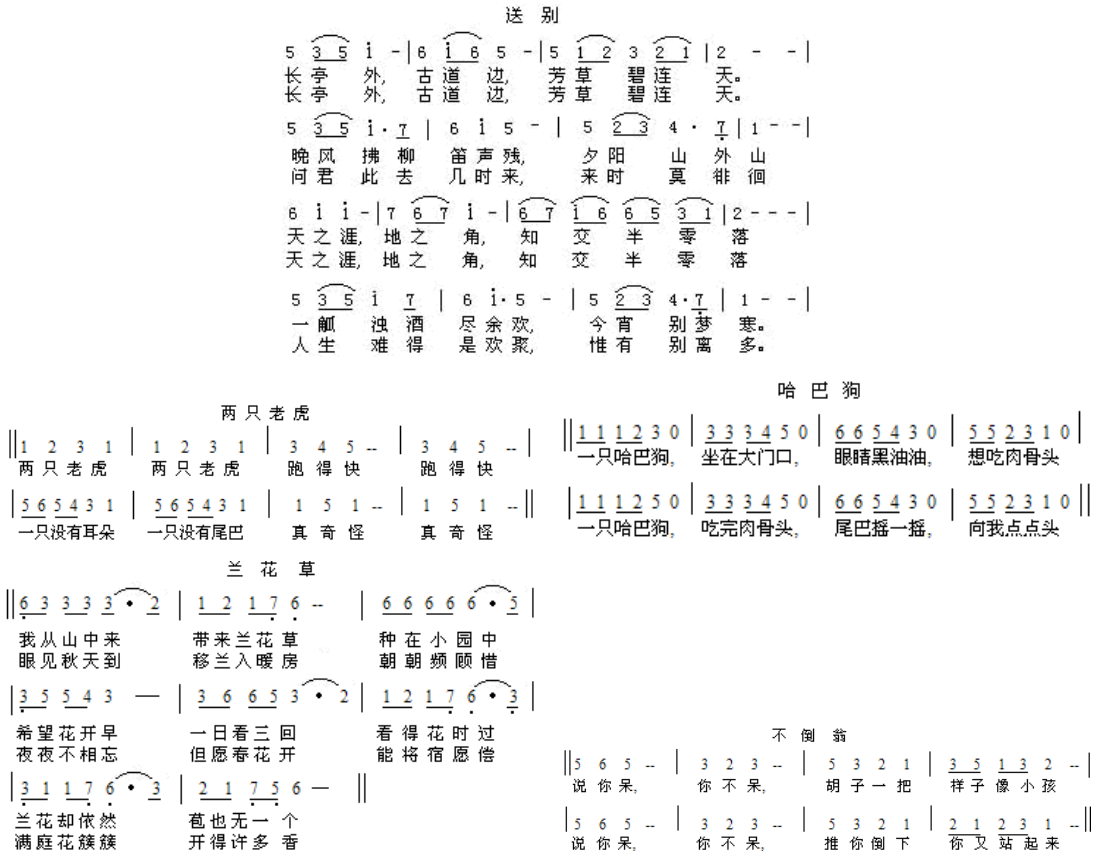


图 15-4　5 首电子音乐的歌曲

源程序

```
/*****************************************************
  File name:          按键选播电子音乐.C
  Chip type:          ATmega16
  Clock frequency:    8.0MHz
*****************************************************/
#include "mega16.h"
#include "delay.h"
```

```
#define uchar unsigned char
#define uint unsigned int
#define   button            PIND.2   //
#define   CS                PORTB.0
#define   LED               PORTC
#define   MIC               PORTD.7
bit     temp;
uchar i;
uchar song_Index=0,Tone_Index=0;          //音乐片段索引，音符索引
uchar *song_Tone,*song_Time;              //音符指针，延时指针
flash uchar tab[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,     //共阳极 LED0～F 的段码
                   0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E,0xBF,0x10};
uint   Tone_tab[]={                                       //1   2   3   4    5   6   7
        50268,51931,53414,54074,55331,56444,57438,        //0   1   2   3    4   5   6
        57887,58721,59465,59804,60433,60990,61487,        //7   8   9   10   11  12  13
        61711,62119,62500,62672,62984,63262,63502,65535};//14  15  16  17   18  19  20
uchar song1_Tone[]={                                      //送别
        11,9,11,14,12,14,12,11,11,7,8,9,8,7,8,            //第 1 行曲子音符
        11,9,11,14,13,12,14,11,11,7,8,9,6,7,              //第 2 行曲子音符
        12,14,14,13,12,13,14,12,13,14,12,12,11,10,7,8,    //第 3 行曲子音符
        11,9,11,14,13,12,14,11,11,8,9,10,6,7,0xff};       //第 4 行曲子音符
uchar song1_Time[]={                                      //节拍
        4,2,2,8,4,2,2,8,4,2,2,4,2,2,12,                   //第 1 行曲子节拍
        4,2,2,4,2,4,4,8,4,2,2,4,2,12,                     //第 2 行曲子节拍
        4,4,8,4,2,2,8,2,2,2,2,2,2,2,2,16,                 //第 3 行曲子节拍
        4,2,2,4,2,4,4,8,4,2,2,4,2,12,0xff};               //第 4 行曲子节拍
uchar song2_Tone[]={                                      //两只老虎
        7,8,9,7,7,8,9,7,9,10,11,9,10,11,
        11,12,11,10,9,7,11,12,11,10,9,7,7,4,7,7,4,7,0xff};
uchar song2_Time[]={
        4,4,4,4,4,4,4,4,4,4,8,4,4,8,
        2,2,2,2,4,4,2,2,2,2,4,4,4,4,8,4,4,8,0xff};
uchar song3_Tone[]={                                      //哈巴狗
        7,7,7,8,9,9,9,9,10,11,12,12,11,10,9,11,11,8,9,7,
        7,7,7,8,11,9,9,9,10,11,12,12,11,10,9,11,11,8,9,7,0xff};
uchar song3_Time[]={
        2,2,2,2,4,2,2,2,2,4,2,2,2,2,4,2,2,2,2,4,
        2,2,2,2,4,2,2,2,2,4,2,2,2,2,4,2,2,2,2,5,0xff};
uchar song4_Tone[]={                                      //兰花草
        5,9,9,9,9,8,7,8,7,6,5,12,12,12,12,12,11,
        2,11,11,10,9,9,12,12,11,9,8,7,8,7,6,5,9,
        2,7,7,6,5,9,8,7,6,4,12,0xff};
uchar song4_Time[]={
        2,2,2,2,2,2,2,2,2,2,8,2,2,2,2,4,2,
        2,2,2,2,8,2,2,2,2,4,2,2,2,2,2,4,2,
        2,2,2,2,4,2,2,2,2,2,8,0xff};
uchar song5_Tone[]={                                      //不倒翁
        11,12,11,9,8,9,11,9,8,7,9,11,7,9,8,
```

```
            11,12,11,9,8,9,11,9,8,7,8,7,8,9,7,0xff};
 uchar song5_Time[]={
              4,4,8,4,4,8,4,4,4,4,2,2,2,2,8,
              4,4,8,4,4,8,4,4,4,4,2,2,2,2,8,0xff};
void delay(uchar k)
{
   uint m,n;
   for(m=0;m<k;m++)
      {
         for(n=0;n<1800;n++)
           {;}
      }
}
void display(void)                                    //LED 数码管显示函数
 {
    CS=1;
    LED=tab[song_Index];
 }
interrupt[EXT_INT0] void int0_isr(void)               //INT0 中断函数
{
   delay(10);
   if(button==0)                                      //再次确认按键，软件防抖
      {
       TIMSK=0x00;
         song_Index++;
      }
   if(song_Index==1)
      {
       song_Tone=song2_Tone;
       song_Time=song2_Time;
      }
   if(song_Index==2)
      {
       song_Tone=song3_Tone;
       song_Time=song3_Time;
      }
   if(song_Index==3)
      {
       song_Tone=song4_Tone;
       song_Time=song4_Time;
      }
   if(song_Index==4)
      {
         song_Tone=song5_Tone;
         song_Time=song5_Time;
      }
   if(song_Index==5)
      {
```

```
            song_Tone=song1_Tone;
            song_Time=song1_Time;
            song_Index=0;
          }
      TIMSK=0x06;
      i=0;
}
interrupt [TIM1_OVF] void timer0_ovf_isr(void)
 {
      TCNT1H=Tone_tab[Tone_Index]/256;;
      TCNT1L=Tone_tab[Tone_Index]%256;
      temp=~temp;
      MIC=temp;
 }
 void port_init(void)
 {
       DDRB=0x01;
      PORTB=0x00;
       DDRC=0xFF;
      PORTC=0xFF;
       DDRD=0xFB;
      PORTD=0x80;
 }
void INT_init(void)                    //中断初始化设置
{
      MCUCR=0x02;                      //定义 INT0 为下降沿时产生中断
       GICR=0x40;                      //允许 INT0 中断
       #asm("sei")                     //开启总中断
}
void TC1_init(void)                    //T/C1 初始化设置
{
      TCCR1A=0x00;                     //WGM13～WGM10=0000，普通模式；COM1A1、
                                       //COM1A0=00，普通端口操作
                                       //COM1B1、COM1B0=00，普通端口操作
      TCCR1B=0x01;                     //CS12～CS10=001，没有预分频
}
 void main(void)
 {
      port_init();
      INT_init();
      TC1_init();
      song_Tone=song1_Tone;
      song_Time=song1_Time;
      while(1)
         {
            display();
            Tone_Index=song_Tone[i];
            if(Tone_Index==0xFF)
```

```
        {
          i=0;
          TIMSK=0x00;
        }
      TIMSK=0x06;
      delay(song_Time[Tone_Index]*60);
      TIMSK=0x00;
      i++;
    }
}
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“按键选播电子音乐.COFF”文件，然后在 Proteus ISIS 中打开已创建的按键选播电子音乐电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试运行状态。在调试运行状态下，LED 数码管显示播放曲子序号值，同时蜂鸣器发出相应的声音，其仿真效果如图 15-5 所示。如果按下按键 K 时，可以选择播放下首歌曲。

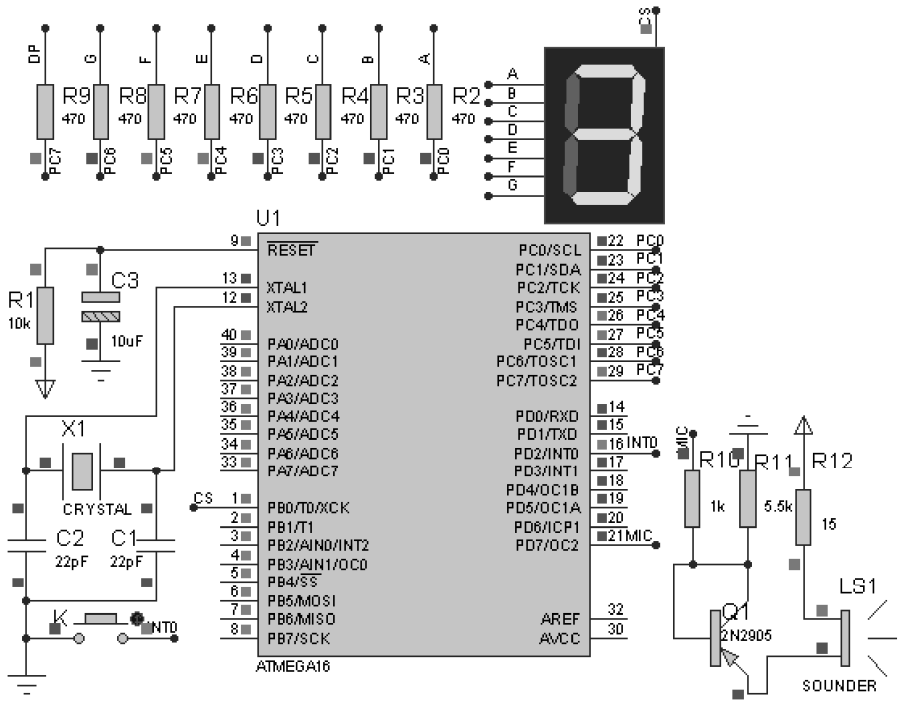


图 15-5　按键选播电子音乐仿真效果图

任务 3　DS1302 时钟日历的设计

设计一个基于 DS1302 芯片的时钟日历，使用字符式 LCD 显示时间和日期。

〖**DS1302 基础知识**〗 DS1302 是美国 DALLAS 公司推出的一种高性能、低功耗的 SPI 总线涓流充电时钟芯片，内含有一个实时时钟/日历和 31B 静态 RAM，通过简单的串行接口与单片机进行通信。实时时钟/日历电路提供秒、分、时、日、日期、月、年的信息，每月的天数和闰年的天数可自动调整，时钟操作可通过 AM/PM 指示决定采用 24h 或 12h 格式。DS1302 与单片机之间能简单地采用同步串行的方式进行通信仅需用到 3 个口线，即 $\overline{\text{RST}}$ 复位）、I/O（数据线）和 SCLK（串行时钟）。时钟/RAM 的读/写数据以 1B 或多达 31B 的字符组方式通信。DS1302 工作时功耗很低，保持数据和时钟信息时功率小于 1mW。DS1302 是由 DS1202 改进而来，增加了以下的特性：双电源引脚用于主电源和备份电源供应 Vcc1。为可编程涓流充电电源，附加 7B 存储器，它广泛应用于电话传真便携式仪器及电池供电的仪器仪表等。

【DS1302 封装形式及引脚说明】 DS1302 包含了 DIP 和 SOIC 两种封装形式，如图 15-6 所示。DS1302 引脚功能如下所述。

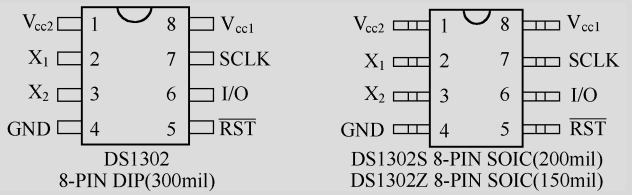


图 15-6 DS1302 封装形式

☺ V_{cc2}：主电源，一般接+5V 电源；
☺ V_{cc1}：辅助电源，一般接 3.6V 可充电池；
☺ X_1 和 X_2：晶振引脚，接 32.768kHz 晶振，通常该引脚上还要接补偿电容；
☺ GND：电源地，接主电源及辅助电源的地端；
☺ SCLK：串行时钟输入端；
☺ I/O：数据 I/O 端；
☺ $\overline{\text{RST}}$：复位输入端。

【DS1302 内部结构及工作原理】 DS1302 的内部结构如图 15-7 所示，它包括输入移位寄存器、控制逻辑、晶振、实时时钟和 31×8 RAM 等部分。

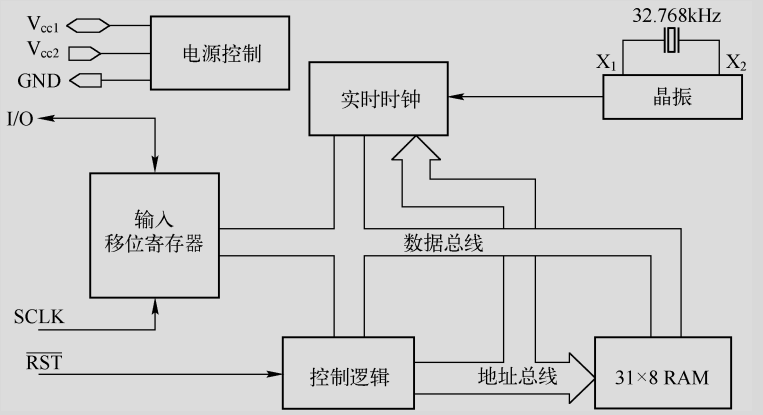


图 15-7 DS1302 内部结构

在进行任何数据传输时，$\overline{RST}$必须被置为高电平（注意，虽然将它置为高电平，内部时钟还是在晶振作用下走时的，此时允许外部读/写数据）。在每个 SCLK 上升沿时数据被输入，下降沿时数据被输出，一次只能读/写一位。是读还是写，需要通过串行输入控制指令来实现（也是一个字节），通过 8 个脉冲便可读取一个字节从而实现串行数据的输入与输出。最初通过 8 个时钟周期载入控制字节到输入移位寄存器。如果控制指令选择的是单字节模式，连续的 8 个时钟脉冲可以进行 8 位数据的写和 8 位数据的读操作。8 个脉冲便可读/写一个字节，SCLK 时钟的上升沿时，数据被写入 DS1302；SCLK 脉冲的下降沿读出 DS1302 的数据。在突发模式，通过连续的脉冲一次性读写完 7B 的时钟/日历寄存器（注意，时钟/日历寄存器要读/写完），也可以一次性读/写 8~32 位 RAM 数据（可按实际情况读/写一定数量的位，不必全部读/写）。

【DS1302 命令字节格式】 每一数据的传送由命令字节进行初始化，DS1302 的命令字节格式见表 15-7，最高位 MSB（D7 位）必须为逻辑 1，如果为 0 则禁止写 DS1302。D6 位为逻辑 0（CLK），指定读/写操作时钟/日历数据；D6 位为逻辑 1（RAM），指定读/写操作为 RAM 数据。D5~D1 位（A4~A1 地址）指定进行输入或输出的特定寄存器。最低有效位 LSB（D0 位）为逻辑 0，指定进行写操作（输入）；为逻辑 1，指定读操作（输出）。命令字节总是从最低有效位 LSB（D0）开始输入，命令字节中的每一位是在 SCLK 的上升沿送出的。

表 15-7　DS1302 的命令字节格式

D7（MSB）	D6	D5	D4	D3	D2	D1	D0（LSB）
1	RAM/$\overline{CLK}$	A4	A3	A2	A1	A0	RD/$\overline{W}$

【复位及时钟控制】 所有的数据传输在$\overline{RST}$置 1 时进行，输入信号有两种功能。首先，$\overline{RST}$接通控制逻辑，允许地址/命令序列送入移位寄存器；其次，$\overline{RST}$提供终止单字节或多字节数据的传送手段。当$\overline{RST}$为高电平时，所有的数据传送被初始化，允许对 DS1302 进行操作。如果在传送过程中$\overline{RST}$置为低电平，则会终止此次数据传送，I/O 引脚变为高阻态。上电运行时，在 V_{cc} 大于或等于 2.5V 前，$\overline{RST}$必须保持低电平。只有在 SCLK 为低电平时，才能将$\overline{RST}$置为高电平。I/O 为串行数据 I/O 端（双向），SCLK 始终是输入端。

【数据传输】 数据的传输主要包括数据输入、数据输出及突发模式，其传输格式如图 15-8 所示。

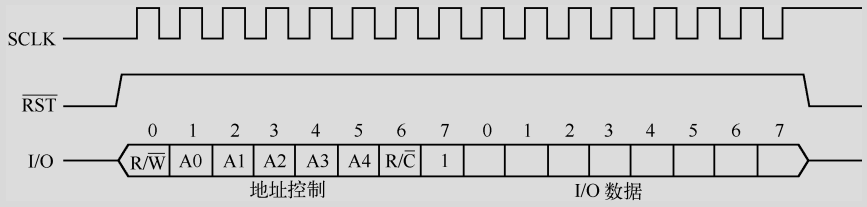
☺ 数据输入：经过 8 个时钟周期的控制字节的输入，一个字节的输入将在下 8 个时钟周期的上升沿完成，数据传输从字节最低位开始。

☺ 数据输出：经过 8 个时钟周期的控制读指令的输入，控制指令串行输入后，一个字节的数据将在下个 8 个时钟周期的下降沿被输出，注意第一位输出是在最后一位控制指令所在脉冲的下降沿被输出，要求$\overline{RST}$保持位高电平。

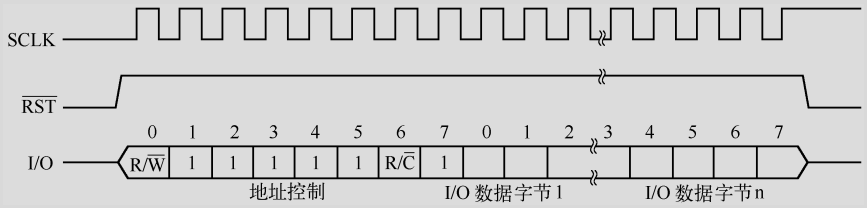
同理，8 个时钟周期的控制读指令如果指定的是突发模式，将会在脉冲的上升沿读入数据，下降沿读出数据，突发模式一次可进行多字节数据的一次性读/写，只要控制好脉冲就行了。

☺ 突发模式：突发模式可以指定为任何时钟/日历或 RAM 的寄存器，与以前一样，

位 6 指定时钟或 RAM，位 0 指定读或写。读取或写入的突发模式开始在位 0 地址 0。



（a）单字节 I/O 数据传输

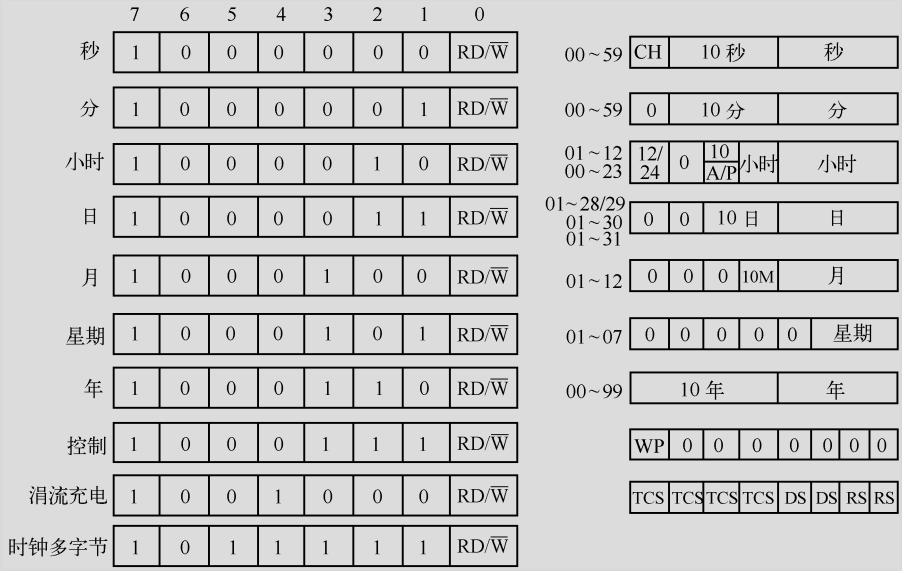


（b）突发模式传输

图 15-8　数据传输

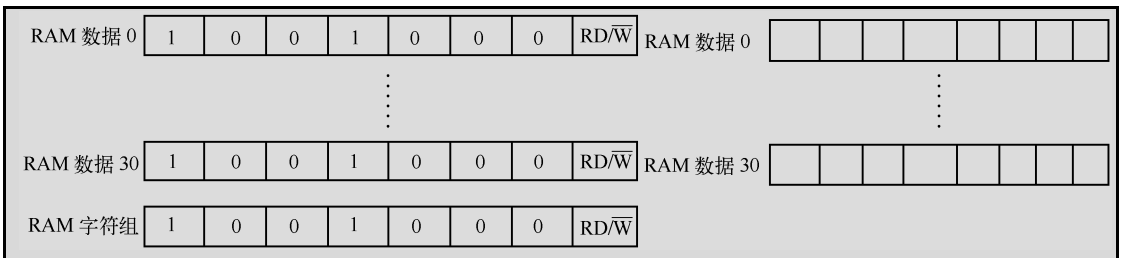
对于 DS1302 来说，在突发模式下写时钟寄存器，起始的 8 个寄存器用来写入相关数据，必须写完。然而，在突发模式下写 RAM 数据时，没有必要全部写完。每个字节都将被写入而不论 31B 是否写完。

【DS1302 内部寄存器】 DS1302 内部寄存器地址（命令）及数据寄存器分配情况如图 15-9 所示。图中 RD/$\overline{W}$ 为读/写保护位：RD/$\overline{W}$ =0，寄存器数据能够写入；RD/$\overline{W}$ =1，寄存器数据不能写入，只能读。A/P=1，下午模式；A/P=0，上午模式。TCS 为涓流充电选择：TCS=1010，使能涓流充电；TCS=其他，禁止涓流充电。DS 为二极管选择位：DS=01 选择一个二极管；DS=10 选择两个二极管；DS=00 或 11 时，即使 TCS=1010，充电功能也被禁止。RS 位功能见表 15-8。



（a）时钟/日历控制部分

图 15-9　DS1302 寄存器分配情况



（b）RAM 控制部分

图 15-9　DS1302 寄存器分配情况（续）

表 15-8　RS 位功能表

RS 位	电　　阻	典型位/kΩ	RS 位	电　　阻	典型位/kΩ
00	无	无	10	R2	4
01	R1	2	11	R3	8



硬件设计

在桌面上双击图标，打开 ISIS 7 Professional 窗口。新建一个 DEFAULT 模板，添加表 15-9 所列的元器件，并完成图 15-10 所示的硬件电路图设计。

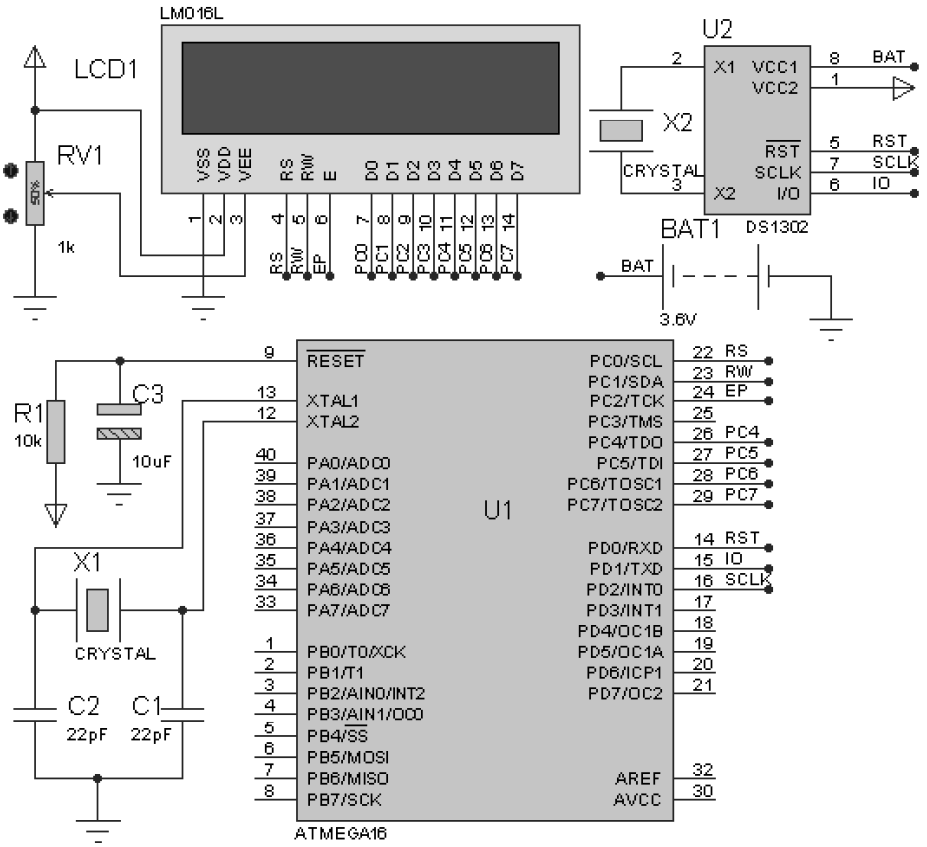


图 15-10　DS1302 时钟日历

表 15-9　DS1302 时钟日历所用元器件

单片机 ATmega16	电解电容 CAP-ELEC 10uF	瓷片电容 CAP22pF	电阻 RES
晶振 CRYSTAL 8MHz	日历时钟芯片 DS1302	可调电阻 POT-HG	LCD 液晶 LM016L

软件设计

在 DS1302 的时钟日历或 RAM 进行数据传送时，DS1302 必须首先发送命令字节。若进行单字节传送，8 位命令字节传送结束后，在下两个 SCLK 周期的上升沿输入数据字节，或者在下 8 个 SCLK 周期的下降沿输出数据字节。DS1302 的实时时间流程如图 15-11 所示，根据该流程可以完成对实时时间的采集。

由于 DS1302 工作在 32.768kHz 的时钟条件下，而单片机工作在 8MHz 时钟环境下，在单片机对 DS1302 进行操作时，由于单片机的工作速度很快，DS1302 可能响应不到，所以在程序中要适当增加一些延时，以使 DS1302 能够正确接收。

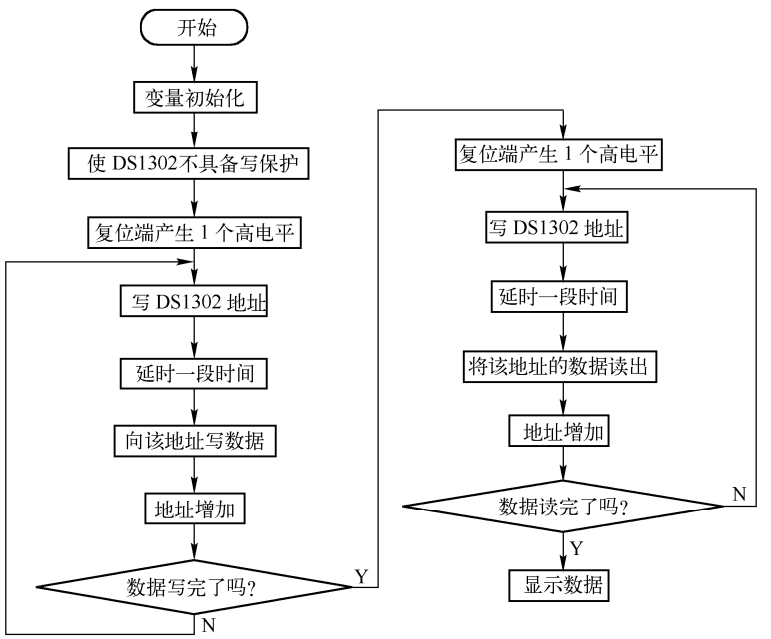


图 15-11　DS1302 实时时间流程图

源程序

```
/*****************************************************
  文件名:          DS1302 时钟日历.c
  单片机型号:      ATmega16
  晶振频率:        8.0MHz
*****************************************************/
#include "mega16.h"
#include "delay.h"
#include    "lcd.h"
#define uchar unsigned char
```

```
#define uint unsigned int
#define DS1302_PORT PORTD
#define DS1302_DDR DDRD
#define DS1302_PIN PIND
#define T_RST 0                                  //PD0
#define T_IO   1                                 //PD1
#define T_CLK 2                                  //PD2
#asm
.equ __lcd_port=0x15;                            //PORTC 数据寄存器地址
#endasm
typedef uchar byte;
#define SETBIT(x,y) (x|=(1<<y))                  //设置指定位为高电平
#define CLRBIT(x,y) (x&=(～(1<<y)))              //设置指定位为低电平
#define CHKBIT(x,y) (x&(1<<y))                   //使能指定位
uchar year=0x13,month=0x02,day=0x14,week=0x04;
uchar hour=0x10,min=0x58,sec=0x12;
uchar temp[7];
uchar time[]="00:00:00";
uchar date[]="00-00-00-0";
void ds1302_write_a_byte(uchar ucDa)             //写一字节数据到 DS1302
{
    uchar i;
    for(i=8; i>0; i--)                           //写 8 位数据
     {
        CLRBIT(DS1302_PORT,T_CLK);               //时钟线清零
        if (ucDa&1)
           {
              SETBIT(DS1302_PORT,T_IO);          //设置 IO 为高
           }
        else
           {
              CLRBIT(DS1302_PORT,T_IO);          //设置 IO 为低电平（允许输入操作）
           }
        SETBIT(DS1302_PORT,T_CLK);               //时钟线为高电平
        ucDa>>=1;                                //每次写一位
     }
}
uchar ds1302_read_a_byte(void)                   //从 DS1302 中读出一字节数据
{
    uchar i,t=0;
    CLRBIT(DS1302_DDR,T_IO);                     //设置 IO 为低电平（允许输入操作）
    CLRBIT(DS1302_PORT,T_IO);                    //IO 线为低电平
    for(i=8; i>0; i--)
       {
          t>>=1;                                 //每次读取 1 位
          SETBIT(DS1302_PORT,T_CLK);
          delay_us(1);
          CLRBIT(DS1302_PORT,T_CLK);
```

```
            delay_us(1);
            if(CHKBIT(DS1302_PIN,T_IO))
               {
                  t|=0x80;
               }
           }
     SETBIT(DS1302_DDR,T_IO);                    //设置 IO 为高电平（允许输出操作）
     return(t);                                  //从 DS1302 中返回读取的数据
}
void write_1302(uchar ucAddr,uchar ucDa)         //写数据到指定地址
{
   CLRBIT(DS1302_PORT,T_RST);
   CLRBIT(DS1302_PORT,T_CLK);
   SETBIT(DS1302_PORT,T_RST);
   ds1302_write_a_byte(ucAddr);
   CLRBIT(DS1302_PORT,T_CLK);
   ds1302_write_a_byte(ucDa);
   CLRBIT(DS1302_PORT,T_CLK);
   CLRBIT(DS1302_PORT,T_RST);
}
uchar read_1302(uchar ucAddr)                    //从指定地址读取数据
{
   uchar ucDa;
   CLRBIT(DS1302_PORT,T_RST);
   CLRBIT(DS1302_PORT,T_CLK);
   SETBIT(DS1302_PORT,T_RST);
   ds1302_write_a_byte(ucAddr);
   ucDa = ds1302_read_a_byte();
   CLRBIT(DS1302_PORT,T_CLK);
   CLRBIT(DS1302_PORT,T_RST);
   return(ucDa);
}

void v_Get1302(uchar ucCurtime[])                //从 DS1302 中获取数据
{
   uchar i;
   uchar ucAddr = 0x81;
   for (i=0;i<7;i++)
      {
         ucCurtime[i]=read_1302(ucAddr);
         ucAddr += 2;
      }
   CLRBIT(DS1302_PORT,T_CLK);
}
void DS1302_init(void)                           //DS1302 初始化
{
   SETBIT(DS1302_PORT,T_CLK);
   SETBIT(DS1302_PORT,T_IO);
```

```
    SETBIT(DS1302_PORT,T_RST);
    SETBIT(DS1302_DDR,T_CLK);
    SETBIT(DS1302_DDR,T_IO);
    SETBIT(DS1302_DDR,T_RST);
    delay_us(1);
    write_1302(0x8e,0x00);
    write_1302(0x90,0xA5);
    write_1302(0x80,0x00);
}
void revise(uchar t[7])
{
    uchar t1,t2,temp,i;
    for(i=0;i<7;i++)
      {
        temp=t[i];
        temp<<=4;
        temp>>=4;
        t1=temp;
        temp=t[i];
        if(i!=2)
            {
              temp>>=4;
            }
        else
            {
              temp<<=1;
              temp>>=5;
            }
        t2=temp;
        t[i]=t1+t2*10;
      }
}
void    lcd_in(void)                          //LCD 显示数据整定
{
      v_Get1302(temp);
      revise(temp);
      time[1]=temp[0]%10+0x30;                //秒个位
      time[2]=(temp[0]/10)%10+0x30;           //秒十位
      time[3]=0x3A;                           //":"
      time[4]=temp[1]%10+0x30;                //分个位
      time[5]=(temp[1]/10)%10+0x30;           //分十位
      time[6]=0x3A;                           //":"
      time[7]=temp[2]%10+0x30;                //时个位
      time[8]=(temp[2]/10)%10+0x30;           //时十位
      date[1]=temp[5]%10+0x30;                //星期
      date[2]=0x2D;                           //"-"
      date[3]=temp[3]%10+0x30;                //天个位
      date[4]=(temp[3]/10)%10+0x30;           //天十位
```

```
    date[5]=0x2D;                          //"-"
    date[6]=temp[4]%10+0x30;               //月个位
    date[7]=(temp[4]/10)%10+0x30;          //月十位
    date[8]=0x2D;                          //"-"
    date[9]=temp[6]%10+0x30;               //年个位
    date[10]=(temp[6]/10)%10+0x30;         //年十位
}
void display1(void)
{
   uint i;
   lcd_in();
   lcd_gotoxy(0,0);
   lcd_putsf("time:");
   lcd_gotoxy(7,0);
   for(i=8;i>0;i--)
      {
        lcd_putchar(time[i]);
      }
}
void display2(void)
{
   uint i;
   lcd_in();
   lcd_gotoxy(0,1);
   lcd_putsf("Date:");
   lcd_gotoxy(6,1);
   for(i=10;i>0;i--)
      {
        lcd_putchar(date[i]);
      }
}
void main(void)
{
  DS1302_init();
  lcd_init(16);
  write_1302(0x8e,0x00);                   //禁止写保护
  write_1302(0x80,sec);                    //秒初始化
  write_1302(0x82,min);                    //分初始化
  write_1302(0x84,hour);                   //时初始化
  write_1302(0x86,day);                    //天初始化
  write_1302(0x88,month);                  //月初始化
  write_1302(0x8a,week);                   //星期初始化
  write_1302(0x8c,year);                   //年初始化
  write_1302(0x8e,0x80);                   //允许写保护
  while(1)
   {
     display1();
     display2();
```

```
    }
}
```

调试与仿真

首先在 CVAVR 中创建项目，输入源代码并生成“DS1302 时钟日历.COFF”文件，然后在 Proteus ISIS 中打开已创建的 DS1302 时钟日历电路图并进行相应设置。单击 Proteus ISIS 模拟调试按钮中的运行按钮，进入调试运行状态。在调试状态下，运行仿真效果如图 15-12 所示。在 DS1302 时钟窗口中，时间的显示格式为：时-分-秒；日期的显示格式为：日-月-年。此系统具有计算 2000 年至 2050 年的秒、分、时、日、月、年、星期的能力。

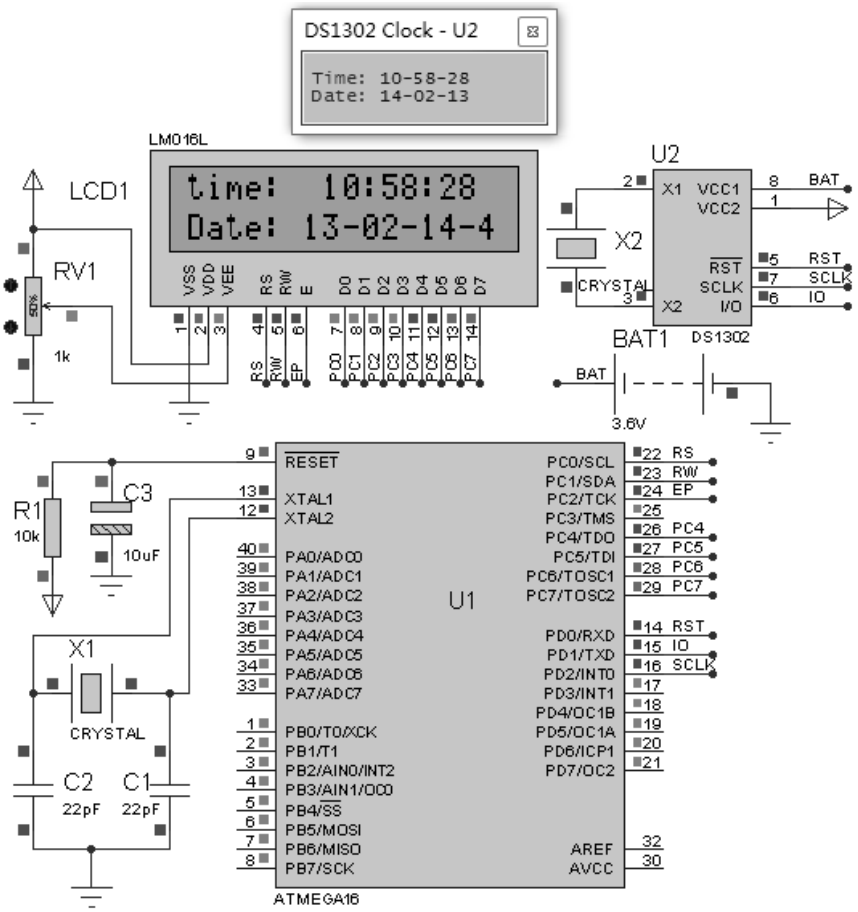


图 15-12　DS1302 时钟日历仿真效果图

附录 A　Proteus 常用快捷键

快　捷　键	功　　能	快　捷　键	功　　能
Ctrl+0	打开设计	Ctrl+Z	撤销
Ctrl+S	保存设计	Ctrl+Y	恢复
R	刷新	E	查找并编辑元器件
G	背景栅格	Ctrl+B	放在后面
O	原点	Ctrl+F	放在前面
X	X 轴指针	Ctrl+N	实时标注
F1	栅格尺寸为 10	W	自动布线
F2	栅格尺寸为 50	T	搜索并标注
F3	栅格尺寸为 100	A	属性分配工具
F4	栅格尺寸为 500	Ctrl+A	网络表导入到 ARES
F5	选择显示中心	Page-up	前一个原理图
F6	缩小	Page-Down	下一个原理图
F7	放大	Alt+X	设计浏览
F8	显示全部	Ctrl+A	增加跟踪曲线
Space	仿真图形	Alt+F12	断点运行
Ctrl+V	查看日记	F10	单步执行
Ctrl+F12	运行/停止调试	F11	跟踪
Pause	暂停运行	Ctrl+F11	单步跳出
Shift+Pause	停止运行	Ctrl+F10	重置弹出窗口
F12	运行	P	选择元器件/符号

附录 B　CodeVisionAVR 编译器常用库函数简介

CodeVisionAVR 编译器中提供了丰富的库函数。

☺ Standard C Input/Output Functions（标准 I/O 函数）
☺ Standard Library Functions（标准库和内存分配函数）
☺ Character Type Functions（字符类型函数）
☺ String Functions（字符串函数）
☺ Mathematical Functions（数学函数）
☺ BCD Conversion Functions（BCD 转换函数）
☺ Gray Code Conversion Functions（格雷码转换函数）
☺ Delay Functions（延时函数）
☺ Memory Access Functions（存储器访问函数）
☺ SPI Functions（SPI 函数）
☺ I^2C Bus Functions（I^2C 总线函数）
☺ 1 Wire Protocol Functions（单总线通信协议函数）
☺ Power Management Functions（电源管理函数）
☺ LCD 函数
　★ LCD Functions（LCD 函数）
　★ LCD Functions for displays with 4x40 characters（4×40 字符型 LCD 函数）
　★ LCD Functions for displays connected in 8 bit memory mapped mode（8 位外部存储器模式接口的 LCD 显示函数）
☺ 温度传感器函数
　★ National Semiconductor LM75 Temperature Sensor Functions（LM75 温度传感器函数）
　★ Dallas Semiconductor DS1621 Thermometer/Thermostat Functions（DS1621 温度传感器函数）
　★ Dallas Semiconductor DS1820/DS1822 Temperature Sensors Functions（DS1820/1822 温度传感器函数）
　★ Dallas Semiconductor DS18B20 Temperature Sensors Functions（DS18B20 温度传感器函数）
☺ 实时时钟函数
　★ Philips PCF8563 Real Time Clock Functions（PCF8563 实时时钟函数）

★ Philips PCF8583 Real Time Clock Functions（PCF8583 实时时钟函数）

★ Dallas Semiconductor DS1302 Real Time Clock Functions（DS1302 实时时钟函数）

★ Dallas Semiconductor DS1307 Real Time Clock Functions（DS1307 实时时钟函数）

☺ E^2PROM 函数

★ DS2430 E^2PROM 函数

★ DS2433 E^2PROM 函数

📖 注意:

使用以上库函数时，必须用“#include”包含相应的头文件。

1）**标准 I/O 函数** 标准 I/O 函数放在“..\INC”目录的“stdio.h”头文件中。

char getchar(void)：使用查询方式返回由 UART 接收的一个字符。

void putchar(char c)：使用查询方式由 UART 发送一个字符 c。

使用这两个函数前，必须设置 UART 的波特率和接收/发送允许。若使用其他的 I/O 外设，必须根据所使用的外设修改 getchar 函数和 putchar 函数。这些函数的源代码在 stdio.h 文件中。所有高级别的 I/O 函数都使用 getchar 函数和 putchar 函数。

void puts(char *str)：使用 putchar 函数把 SRAM 中的以空字符结束的字符串输出，并在后面加换行符。

void putsf(char flash *str)：使用 putchar 函数把 FLASH 中的以空字符结束的字符串输出，并在后面加换行符。

void printf(char flash *fmtstr [, arg1, arg2, ...])：使用 putchar 函数按格式说明符输出格式化文本 fmtstr 字符串。格式化文本 fmtstr 字符串是常量，必须放在 FLASH 中。

printf 执行的是标准 C 的一个子集，其格式化说明符如下所述。

%c: 输出一个 ASCII 字符。

%d: 输出有符号十进制整数。

%i: 输出有符号十进制整数。

%u: 输出无符号十进制整数。

%x: 输出小写字母的十六进制整数。

%X: 输出大写字母的十六进制整数。

%s: 输出 SRAM 中的以空字符结束的字符串。

%%: 输出“%”字符。

所有输出的数都是右对齐的，并在左侧加空格补齐。如果在“%”与“d”、“i”、“u”、“x”或“X”之间加入一个字符 0，则输出的数的左侧加 0 补齐；如果在“%”与“d”、“i”、“u”、“x”或“X”之间加入一个字符“-”，则输出的数左对齐。

void sprintf(char *str, char flash *fmtstr [, arg1, arg2, ...])：这个函数与 printf 类似，只是它的格式化字符放在以空字符结尾的字符串 str 中。

char *gets(char *str, unsigned char len)：使用 getchar 函数接收以换行符结束的字符串 str。换行符会被 0 替换。字符串的最大长度是 len。如果已经收到了 len 个字符后仍未收到换行符，则字符串就以 0 结束，函数停止执行并退出。函数的返回值是指向 str 的指针。

signed char scanf(char flash *fmtstr [, arg1 address, arg2 address, ...])：使用 getchar 函数按格式说明符接收格式化文本 fmtstr 字符串。格式化文本 fmtstr 字符串是常量，必须放在 FLASH 中。

scanf 执行的是标准 C 的一个子集，其格式化说明符如下所述。
%c：接收一个 ASCII 字符。
%d：接收有符号十进制整数。
%i：接收有符号十进制整数。
%u：接收无符号十进制整数。
%x：接收无符号十六进制整数。
%s：接收以空字符结束的字符串。
函数返回成功接收的字符个数，如果返回−1 表示接收出错。

signed char sscanf(char *str, char flash *fmtstr [, arg1 address, arg2 address, ...])：这个函数与 scanf 类似，只是它的格式化字符放在 SRAM 中的以空字符结尾的字符串 str 中。

2）**标准库和内存分配函数**　这些函数放在“..\INC”目录的“stdio.h”头文件中。

int atoi(char *str)：转换字符串 str 为整型数并返回其值，字符串 str 起始必须是十进制数字的字符，否则返回 0。当遇到字符串中第一个非十进制数字的字符时，转换结束。

long int atol(char *str))：转换字符串 str 为长整型数并返回其值，字符串 str 的起始必须是长整型数形式的字符，否则返回 0。

void itoa(int n, char *str)：转换整型数 *n* 为字符串 str。

void ltoa(long int n, char *str)：转换长整型数 *n* 为字符串 str。

void ftoa(float n, unsigned char decimals, char *str)：转换浮点数 *n* 为字符串 str，由 decimals 指定四舍五入保留的小数位（最多 5 位）。

void ftoe(float n, unsigned char decimals, char *str)：转换浮点数 *n* 为字符串 str。字符串表示为科学计数法形式，由 decimals 指定四舍五入保留的小数位（最多 5 位）。

float atof(char *str)：转换字符串 str 为浮点数并返回其值，字符串 str 起始必须是数字或小数点，否则返回 0。当碰到字符串中第一个十进制数字和小数点外的字符时，转换结束。

int rand (void)：产生一个 0～32767 之间的伪随机数。

void srand(int seed)：设置伪随机数发生器的种子数。

3）**字符类型函数**　这些函数支持 ASCII 字符运算，放在“..\INC”目录的“ctype.h”头文件中。

unsigned char isalnum(char c)：如果 c 是数字或字母则返回 1，否则返回 0。

unsigned char isalpha(char c)：如果 c 是字母则返回 1，否则返回 0。

unsigned char isascii(char c)：如果 c 是 ASCII 码（0～127）则返回 1，否则返回 0。

unsigned char iscntrl(char c)：如果 c 是控制字符（0～31 或 127）则返回 1，否则返回 0。

unsigned char isdigit(char c)：如果 c 是十进制数字则返回 1，否则返回 0。

unsigned char islower(char c)：如果 c 是小写字母则返回 1，否则返回 0。

unsigned char isphnt(char c)：如果 c 是一个可打印字符（32～127）则返回 1，否则返回 0。

unsigned char ispunct(char c)：如果 c 是一个标点字符（ASCII 码中除了控制字符、数字和字母外的字符）则返回 1，否则返回 0。

unsigned char isspace(char c)：如果 c 是空格则返回 1，否则返回 0。

unsigned char isupper(char c)：如果 c 是大写字母则返回 1，否则返回 0。

unsigned char isxdigit(char c)：如果 c 是十六进制数字则返回 1，否则返回 0。

chartoascii(char c)：返回字符 c 对应的 ASCII。

unsigned char toint(char c)：把 c 当做十六进制字符并返回对应的十进制整数（0～15）。

char tolower(char c)：如果 c 是大写字母则返回对应的小写字母，其他字符保持不变。

char toupper(char c)：如果 c 是小写字母则返回对应的大写字母，其他字符保持不变。

4）**字符串函数** 字符串函数用于 SRAM 和 FLASH 中的字符串的操作，这些函数放在“..\INC”目录的“string.h”头文件中。

char *strcat(char *str1, char *str2)：复制字符串 str2 到字符串 str1 的结尾，返回 str1 的指针。

char *strcatf(char *str1, char flash *str2)：复制 FLASH 中的 str2 到 str1 的结尾，返回 str1 的指针。

char *strncat(char *str1, char *str2, unsigned char n)：复制 str2（不含结束符 NULL）的 *n* 个字符到 str1 的结尾。如果 str2 的长度比 *n* 小，则只复制 str2，返回 str1 的指针。

char *strncatf(char *str1, char flash *str2, unsigned char n)：复制 FLASH 中的字符串 str2（不含结束符 NULL）的 *n* 个字符到 str1 的结尾。如果 str2 的长度比 *n* 小，则只复制 str2，返回 str1 的指针。

char *strchr(char *str, char c)：在字符串 str 中搜索第一个出现的字符 c。如果成功，则返回匹配字符的指针；如果没有搜索到匹配字符，则返回 NULL。

char *strrchr(char *str, char c)：在字符串 str 中搜索最后一个出现的字符 c。如果成功，则返回匹配字符的指针；如果没有搜索到匹配字符，则返回 NULL。

signed char strpos(char *str, char c)：在字符串 str 中搜索第一个出现的字符 c。如果成功，则返回匹配字符在字符串中的位置；如果没有搜索到匹配字符，则返回–1。

signed char strrpos(char *str, char c)：在字符串 str 中搜索最后一个出现的字符 c。如果成功，则返回匹配字符在字符串中的位置；如果没有搜索到匹配字符，则返回–1。

signed char strcmp(char *str1, char *str2)：比较两个字符串。如果相同，则返回 0；如果 str1>str2，则返回值>0；如果 str1<str2，则返回值<0。

signed char strcmpf(char *str1, char flash *str2)：比较 SRAM 中的字符串 str1 和 FLASH 中的字符串 str2。如果相同，则返回 0；如果 str1>str2，则返回值>0；如果 str1<str2，则返回值<0。

signed char strncmp(char *str1, char *str2, unsigned char n)：比较两个字符串的前 *n* 个字符。如果相同，则返回 0；如果 str1>str2，则返回值>0；如果 str1<str2，则返回值<0。

signed char strncmpf(char *str1, char flash *str2, unsigned char n)：比较 SRAM 中的字符串 str1 和 FLASH 中的字符串 str2 的前 *n* 个字符。如果相同，则返回 0；如果 str1>str2，则返回值>0；如果 str1<str2，则返回值<0。

char *strcpy(char *dest, char *src)：复制字符串 src 到字符串 dest 中，则返回 dest 的指针。

char *strcpyf(char *dest, char flash *src)：复制 FLASH 中的字符串 src 到 SRAM 中的字符串 dest 中，则返回 dest 的指针。

char *strncpy(char *dest, char *src, unsigned char n)：复制字符串 src 的前 *n* 个字符到字符串 dest 中，返回 dest 的指针。

char *strncpyf(char *dest, char flash *src, unsigned char n)：复制 FLASH 中的字符串 src 的前 *n* 个字符到 SRAM 中的字符串 dest 中，返回 dest 的指针。

unsigned char strspn(char *str, char *set)：在字符串 str 中搜索与字符串 set 不匹配的第一个字符。如果搜索到不匹配，则返回不匹配字符在 str 的位置；如果 set 的所有字符都匹配，则返回字符串 str 的长度。

unsigned char strspnf(char *str, char flash *set)：在 SRAM 中的字符串 str 中搜索与 FLASH 中的字符串 set 不匹配的第一个字符。如果搜索到不匹配，则返回不匹配字符在 str 的位置；如果 set 的所有字符都匹配，则返回字符串 str 的长度。

unsigned char strcspn(char *str, char *set)：在字符串 str 中搜索与字符串 set 匹配的第一个字符。如果搜索到匹配，则返回匹配字符在 str 的位置；如果没有匹配字符，则返回字符串 str 的长度。

unsigned char strcspnf(char *str, char flash *set)：在字符串 str 中搜索与 FLASH 中的字符串 set 匹配的第一个字符。如果搜索到匹配，则返回匹配字符在 str 的位置；如果没有匹配字符，则返回字符串 str 的长度。

char *strpbrk(char *str, char *set)：在字符串 str 中搜索与字符串 set 匹配的第一个字符。如果搜索到匹配，则返回匹配字符的指针；如果没有匹配字符，则返回 NULL。

char *strpbrkf(char *str, char flash *set)：在字符串 str 中搜索与 FLASH 中的字符串 set 匹配的第一个字符。如果搜索到匹配，则返回匹配字符的指针；如果没有匹配字符，则返回 NULL。

char *strrpbrk(char *str, char *set)：在字符串 str 中搜索与字符串 set 匹配的最后一个字符。如果搜索到匹配，则返回匹配字符的指针；如果没有匹配字符，则返回 NULL。

char *strrpbrkf(char *str, char flash *set)：在 SRAM 中的字符串 str 中搜索与 FLASH 中的字符串 set 匹配的最后一个字符。如果搜索到匹配，则返回匹配字符的指针；如果没有匹配字符，则返回 NULL。

char *strstr(char *str1, char *str2)：在字符串 str1 中搜索与字符串 str2 匹配的子字符串。如果找到匹配的子字符串，则返回 str1 中的子字符串的起始地址指针；否则返回 NULL。

char *strstrf(char *str1, char flash *str2)：在 SRAM 中的字符串 str1 中搜索与 FLASH 中的字符串 str2 匹配的子字符串。如果找到匹配的子字符串，则返回 str1 中的子字符串的起始地址指针；否则返回 NULL。

unsigned char strlen(char *str)：返回字符串 str 的长度（0～255）。

unsigned int _strlen(char *str)：返回字符串 str 的长度（0～65535）。这个函数只能用在 SMALL 模式下。

unsigned int strlenf(char flash *str)：返回 FLASH 中的字符串 str 的长度。

void *memcpy(void *dest,void *src，unsigned char n)：复制 src 的 *n* 个字节到 dest 中。dest 与 src 不能重叠。返回 dest 的指针。这个函数只能用在 TINY 模式下。

void *memcpy(void *dest,void *src，unsigned int n)：复制 src 的 *n* 个字节到 dest 中。dest 与 src 不能重叠。返回 dest 的指针。这个函数只能用在 SMALL 模式下。

void *memcpyf(void *dest,void flash *src，unsigned char n)：复制 Flash 中的字符串 src 的 *n* 个字节到 dost 中。dest 与 src 不能重叠。返回 dest 的指针。这个函数只能用在 TINY 模式下。

void *memcpy(void *dest,void flash *src，unsigned int n)：复制 Flash 中的字符串 src 的 *n* 个字节到 dest 中。dest 与 src 不能重叠。返回 dest 的指针。这个函数只能用在 SMALL 模式下。

void *memccpy(void *dest,void *src，char c，unsigned char n)：复制字符串 src 的 *n* 个字节到 dest 中，如果碰到字符 c 就停止。dest 与 src 不能重叠。如果最后一个复制的字符是 c，则返回 NULL，否则返回指向 dest+n+1 的指针。这个函数只能用在 TINY 模式下。

void *memcepy(void *dest,void *src，char c，unsignedhtn)：复制字符串 src 的 *n* 个字节到 dest 中，如果碰到字符 c 就停止。dest 与 src 不能重叠。如果最后一个复制的字符是 c，则返回 NULL，否则返回指向 dest+n+1 的指针。这个函数只能用在 SMALL 模式下。

void *memmove(void *dest,void *src，unsigned char n)：复制 src 的 *n* 个字节到 dest 中。dest 与 src 可以重叠。返回 dest 的指针。这个函数只能用在 TINY 模式下。

void *memmove(void *dest,void *src，unsigned int n)：复制 src 的 *n* 个字节到 dest 中。dest 与 src 可以重叠。返回 dest 的指针。这个函数只能用在 SMALL 模式下。

void *memchr(void *buf，unsigned char c，unsigned char n)：在 buf 的前 *n* 个字节中搜索字符 c，如果搜索到 c，就返回指向 c 的指针；否则返回 NULL。这个函数只能用在 TINY 模式下。

void *memchr(void *buf，unsigned char c，unsigned int n)：在 buf 的前 *n* 个字节中搜索字符 c，如果搜索到 c，就返回指向 c 的指针；否则返回 NULL。这个函数只能用在 SMALL 模式下。

signed char memcmp(void *buf1,void *buf2，unsigned char n)：比较字符串 buf1 和 buf2 的前 *n* 个字节。当 buf1<buf2，buf1=buf2，buf1>buf2 时，分别返回<0，0，>0。这个函数只能用在 TINY 模式下。

signed char memcmp(void *buf1,void *buf2，unsigned int n)：比较字符串 buf1 和 buf2 的前 *n* 个字节。当 buf1<buf2，buf1=buf2，buf1>buf2 时，分别返回<0，0，>0。这个函数只能用在 SMALL 模式下。

signed char memcmpf(void *buf1,voidflash *buf2，unsigned char n)：比较 SRAM 中的字符串 buf1 和 FLASH 中的字符串 buf2，最多比较前 *n* 个字节。当 bufl<buf2，buf1=buf2，bufl>buf2 时，分别返回<0，0，>0。这个函数只能用在 TINY 模式下。

signed char memcmpf(void *buf1,voidflash *buf2，unsigned int n)：比较 SRAM 中的字符串 buf1 和 FLASH 中的字符串 buf2，最多比较前 *n* 个字节。当 buf1<buf2，buf1=buf2，buf1>buf2 时，分别返回<0，0，>0。这个函数只能用在 SMALL 模式下。

void *memstr(void *buf，unsigned char c，unsigned char n)：用字符 c 填充 buf 的前 *n* 个字节，返回指向 bur 的指针。这个函数只能用在 TINY 模式下。

void *memstr(void *buf，unsigned char c，unsigned int n)：用字符 c 填充 buf 的前 *n* 个字节，返回指向 buf 的指针。这个函数只能用在 SMALL 模式下。

5）**数学函数** 这些函数的原型放在“..\INC”目录的“math.h”头文件中。

unsigned char cabs(char n)：返回 n 的绝对值。

unsigned int abs(char n)：返回 n 的绝对值。

unsigned long labs(long int n)：返回 n 的绝对值。

float　fabs(float n)：返回 n 的绝对值。

signed char cmax(char a,char b)：返回 a 和 b 的最大值。

int max(inta，int b)：返回 a 和 b 的最大值。

long int lmax(long int a, long int b)：返回 a 和 b 的最大值

float　fman(float　a,float b)：返回 a 和 b 的最大值。

char cmin(char a,char b)：返回 a 和 b 的最小值。

int min(int a，int b)：返回 a 和 b 的最小值。

long int lmin(long int a，long int b)：返回 a 和 b 的最小值。

float fmin(float a，float b)：返回 a 和 b 的最小值。

char csign(char n)：当 n 分别为负数、0、正数时，返回−1、0、1。

char sign(int n)：当 n 分别为负数、0、正数时，返回−1、0、1。

char lsign(long int n)：当 n 分别为负数、0、正数时，返回−1、0、1。

char fsign(float n)：当 n 分别为负数、0、正数时，返回−1、0、1。

unsigned char sqrt(unsigned int n)：返回无符号整数 n 的平方根。

unsigned int lsqrt(unsigned long n)：返回无符号长整数 n 的平方根。

float sqrt(float n)：返回浮点数 n 的平方根。

float floor(float n)：返回不大于 n 的最大整数。

float ceil(float n)：返回对应 n 的整数，小数部分四舍五入。

float fmod(float n，float y)：返回 r/y 的余数。

float modf(float n，float *ipart)：把浮点数 n 分解成整数部分和小数部分。整数部分存放在 ipart 指向的变量中，小数部分应大于或等于 0.5 而小于 1，并作为函数的返回值。

float ldexp(float n，int expn)：返回 $n\times2^{e\times pn}$。

float frexp(float n，int *expn)：把浮点数 n 分解成数字部分 y（尾数）和以 2 为底的指数 expn 两个部分，即 $n=y\times2^{expn}$，y 要大于等于 0.5 且小于 1，y 值被函数返回，而 expn 值存放在 expn 指向的变量中。

float exp(float n)：返回 e^n 的值。

float log(float n)：返回 n 的自然对数。

float logl0(float n)：返回以 10 为底的 n 的对数。

float pow(float n，float y)：返回 n^y 的值。

float sin(float n)：返回 n 的正弦函数值，n 为弧度。

float cos(float n)：返回 n 的余弦函数值，n 为弧度。

float tan(float n)：返回 n 的正切函数值，n 为弧度。

float sinh(float n)：返回 n 的双曲正弦函数值，n 为弧度。

float cosh(float n)：返回 n 的双曲余弦函数值，n 为弧度。

float tanh(float n)：返回 n 的双曲正切函数值，n 为弧度。

float asin(float n)：返回 n 的反正弦函数值，返回值为弧度，范围在−π/2～π/2 之间，n 的值必须在−1～1 之间。

float acos(float n)：返回 n 的反余弦函数值，返回值为弧度，范围在 0～π/2 之间，n 的

值必须在−1～1 之间。

float atan(float n)：返回 n 的反正弦函数值，返回值为弧度，范围在−π/2～π/2 之间。

float atall2(float y，float n)：返回 y/n 的反正弦函数值，返回值为弧度，范围在−π～π之间。

6）**BCD 转换函数** 这些函数放在“..\INC”目录的“bcd.h”头文件中。

unsigned char bcd2bin(unsigned char n)：把 BCD 码数 n 转换为二进制。

unsigned char bin2bcd(unsigned char n)：把二进制数 n 转换为 BCD 码。n 必须在 0～99之间。

7）**格雷码转换函数** 这些函数放在“..\INC”目录的“gray.h”头文件中。

将格雷码形式的 n 转换到二进制形式的函数有：

```
unsigned char gray2binc(unsigned char n);
unsigned char gray2bin(unsigned int n);
unsigned char gray2binl(unsigned long n)。
```

将二进制形式的 n 转换到格雷码形式的函数有：

```
unsigned char bin2grayc(unsigned char n);
unsigned char bin2gray(unsigned int n);
unsigned char bin2grayl(unsigned long n)。
```

8）**延时函数** 延时函数使用程序循环产生，调用这些函数前要关闭中断，否则会比预期的延时要长。同时一定要在“Project”→“Configure”→“C Compiler”菜单中设定正确的时钟频率。这些函数的原型放在“..\INC”目录的“delay.h”头文件中。

void delay_us(unsigned int n)：n 个微秒的延时。n 必须是常数表达式。

void delay_ms(unsigned int n)：n 个毫秒的延时。n 必须是常数表达式。这个函数会每1ms 清一次看门狗。

9）**存储器访问函数** 这些函数放在“..\INC”目录的“mem.h”头文件中。

void pokeb(unsigned int addr,unsigned char data)：把 1Bdata 写在 SRAM 中指定的 addr 地址。

void pokew(unsigned int addr,unsigned int data)：把 1Wdata 写在 SRAM 中指定的 addr 地址。低字节在 addr，高字节在 addr+1。

unsigned char peekb(unsigned int addr)：在 SRAM 中指定的 addr 地址读 1B。

unsigned int peekw(unsigned int addr)：在 SRAM 中指定的 addr 地址读 1W。

10）**SPI 函数** 这些函数放在“..\INC”目录的“spi.h”头文件中。

unsigned char spi(unsigned char data)：发送 1B，同时接收 1B。

调用 SPI 函数前要先设置 SPI 控制寄存器 SPCR。由于 SPI 函数通信使用查询方式，所以不需要设置 SPI 中断允许标志位 SPIE。

11）**I^2C 总线函数** I^2C 总线函数可以将单片机作为主机或从机。这些函数放在“..\INC”目录的“i2c.h”头文件中。使用这些函数前必须用“#include”，并要声明单片机的哪些端口线用于 I^2C 总线。

void i2c_init(void)：初始化 I^2C 总线。调用其他 I^2C 总线函数前必须先调用此函数。

unsigned char i2c_start(void)：发送 START 信号。如果总线空闲，则返回 1；如果总线

忙，则返回 0。

void i2c_stop(void)：发送 STOP 信号。

unsigned char i2c_read(unsigned char ack)：读 1B。参数定义了在 1B 被读取后是否需要发送请求。可以置 ACK 为 0 或 1。

unsigned char i2c_write(unsigned char data)：写 1B。如果从机有应答，则返回 1；如果从机不应答，则返回 0。

12）**单总线通信协议函数**　只有商业版的 CodeVisionAVR C Compiler 才有这部分功能。这些函数放在“..\INC”目录的“1wire.h”头文件中。

这些函数以 MCU 为主机，外设（单线总线器件）为从机。包含头文件前，用户必须先声明哪些口线使用单线通信协议与器件通信。

由于单总线协议函数使用时有严格的延时，所以操作期间要关闭中断。并且一定要执行菜单命令“Project”→“Configure”→“C Compiler”来设定正确的晶振频率。

unsigned char w1_init(void)：初始化总线上的器件。如果有，则器件返回 1，否则返回 0。

unsigned char w1_read(void)：从总线上读 1B。

unsigned char w1_write(unsigned char data)：在总线上写 1B。如果写过程正常完成，则返回 1，否则返回 0。

unsigned char w1_search(unsigned char cmd,void *p)：返回总线上器件的个数。如果没有器件，则返回 0。参数 cmd 是发给器件的命令，如 DS1820/DS1822 的搜索 ROM（Search ROM）命令字为 F0h，报警搜索（Alarm Search）命令字为 Ech。指针 p 指向存放器件返回的 8B ROM 码的 SRAM 区域。8B 后会存放某些器件的状态字节，如 DS2405。因此，必须给每个器件开辟 9B 的 SRAM。

unsigned char w1_crc8(void *p, unsigned char n)：返回从地址 p 开始的 n 个字节的 8 位 CRC 校验。

13）**电源管理函数**　这些函数放在“..\INC”目录的“sleep.h”头文件中。

void sleep_enable(void)：允许进入低功耗模式。

void sleep_disable(void)：禁止进入低功耗模式。这可以防止意外进入低功耗模式。

void idle(void)：进入空闲模式。调用这个函数前，必须先调用 sleep_enable 函数，以允许进入低功耗模式。在这种模式下，CPU 停止工作，但定时器、看门狗和中断系统继续工作。MCU 可以由使能的中断唤醒。

void powerdown(void)：进入掉电模式。调用这个函数前，必须先调用 sleep_enable 函数，以允许进入低功耗模式。在这种模式下，外部晶振停振，看门狗和外部中断继续工作。只有外部复位、看门狗复位和外部中断可以唤醒 MCU。

void powersave(void)：进入省电模式。调用这个函数前，必须先调用 sleep_enable 函数，以允许进入低功耗模式。这种模式与掉电模式类似，但略有不同。

14）**LCD 函数**

（1）LCD Functions（LCD 函数）：使用这些函数前，应当用#include“lcd.H”预处理。在包含头文件前，必须声明哪一个口与 LCD 模块通信。下面的函数针对由日立 HD44780 或兼容芯片控制的字符型 LCD 模块，支持以下类型：1×8, 2×12, 3×12, 1×16, 2×16, 2×20, 4×20, 2×24, 2×40。

【底层 LCD 函数】

void_lcd ready(void)：等待，直到 LCD 模块准备好接收数据为止。在使用_lcd_write data 函数向 LCD 模块写数据前必须调用此函数。

void_lcd_write_data(unsigned char data)：向 LCD 模块的命令寄存器写 1B data。

voidlcd_write_byte(unsigned char addr，unsigned char data)：向 LCD 模块的字符发生器或 RAM 中写 1B data。

unsigned char lcd_read_byte(unsigned char addr)：从 LCD 模块的字符发生器或 RAM 读出 1B。

【高级 LCD 函数】

void lcd_init(unsigned char lcd_columns)：初始化 LCD 模块，清屏并把显示坐标设定在 0 列 0 行，模块不显示光标，LCD 模块的列必须指定（如 16），在使用其他高级 LCD 函数前必须先调用此函数。

void lcd_clear(void)：清屏并把显示坐标设定在 0 列 0 行。

void lcd_gotoxy(unsigned char x，unsigned char y)：设定显示坐标在 x 列 y 行。列、行均从 0 开始。

void lcd_putchar (char c)：在当前坐标显示字符 c。

void lcd_puts(char *str)：在当前坐标显示 SRAM 中的字符串 str。

void lcd_putsf(char flash *str)：在当前坐标显示 FLASH 中的字符串 str。

（2）LCD Functions for displays with 4x40 characters（4×40 字符型 LCD 函数）：只有商业版的 CodeVisionAVR C Compiler 才有这部分功能。这些 LCD 函数针对由日立 HD44780 或兼容芯片控制的字符型 4×40 LCD 模块。其函数放在“..\INC”目录的“lcd4x40.h”头文件中。包含头文件前，必须声明哪一个口与 LCD 模块通信。

【底层 LCD 函数】

void _lcd_ready(void)：等待，直到 LCD 模块准备好接收数据为止。在使用_lcd_write _data 函数向 LCD 模块写数据前，必须调用此函数。

void _lcd_write_data(unsigned char data)：向 LCD 模块的命令寄存器写 1B data。这个函数可以用于修改 LCD 的配置。

在调用_lcd_ready 和_lcd_write_data 函数前，全局变量_en1_msk 必须设为 LCD_EN1（LCD_EN2），以选择使用上半部（下半部）的 LCD 控制器。

void lcd_write_byte(unsigned char addr, unsigned char data)：向 LCD 模块的字符发生器或显存写 1B data。

unsigned char lcd_read_byte(unsigned char addr)：从 LCD 模块的字符发生器或显存读出 1B。

【高级 LCD 函数】

void lcd_init(void)：初始化 LCD 模块，清屏并把显示坐标设定在 0 列 0 行。LCD 模块的列必须指定（如 14）。这时不显示光标。在使用其他高级 LCD 函数前，必须先调用此函数。

void lcd_clear(void)：清屏并把显示坐标设定在 0 列 0 行。

void lcd_gotoxy(unsigned char x, unsigned char y)：设定显示坐标在 x 列 y 行。列、行从 0 开始。

void lcd_putchar(char c)：在当前坐标显示字符 c。

void lcd_puts(char *str)：在当前坐标显示 SRAM 中的字符串 str。

void lcd_putsf(char flash *str)：在当前坐标显示 FLASH 中的字符串 str。

（3）LCD Functions for displays connected in 8 bit memory mapped mode（以 8 位外部存储器模式接口的 LCD 显示函数）：这些 LCD 函数针对的是由日立 HD44780 或兼容芯片控制的字符型 LCD 模块。LCD 作为一个 8 位的外设接在 AVR 的外部数据地址总线上。这种接口方式可以用在 STK200 和 STK300 开发板上。LCD 的连接方式参考开发板的说明。这些函数只能用于能外扩存储器的 AVR 芯片：AT90S4414, AT90S 8515, ATmega603, Atmega 103 和 ATmega161。

这些函数放在“..\INC”目录的“lcdstk.h”头文件中。在 lcdstk.h 中，支持以下类型的字符行 LCD 模块：1×8, 2×12, 3×12, 1×16, 2×16, 2×20, 4×20, 2×24, 2×40。

【底层 LCD 函数】

void _lcd_ready(void)：等待，直到 LCD 模块准备好接收数据为止。在使用宏_LCD_RS0 和_LCD_RS1_向 LCD 模块写数据前必须调用此函数。宏_LCD_RS0（或_LCD_RS1）用来设 RS=0（或 RS=1）并访问 LCD 指令寄存器。

void lcd_write_byte(unsigned char addr, unsigned char data)：向 LCD 的字符发生器或显存写 1B。

unsigned char lcd_read_byte(unsigned char addr)：从 LCD 模块的字符发生器或显存读出 1B。

【高级 LCD 函数】

void lcd_init(unsigned char lcd_columns)：初始化 LCD 模块，清屏并把显示坐标设定在 0 列 0 行。LCD 模块的列必须指定（如 12）。这时不显示光标。在使用其他高级 LCD 函数前，必须先调用此函数。

void lcd_clear(void)：清屏并把显示坐标设定在 0 列 0 行。

void lcd_gotoxy(unsigned char x, unsigned char y)：设定显示坐标在 x 列 y 行。列、行从 0 开始。

void lcd_putchar(char c)：在当前坐标显示字符 c。

void lcd_puts(char *str)：在当前坐标显示 SRAM 中的字符串 str。

void lcd_putsf(char flash *str)：在当前坐标显示 FLASH 中的字符串 str。

15）温度传感函数

（1）National Semiconductor LM75 Temperature Sensor Functions（LM75 温度传感器函数）：这些函数放在“..\INC”目录的“lm75.h”头文件中。I^2C 总线函数在 lm75.h 中自动包含。包含头文件前，必须先声明哪些口线通过 I^2C 总线与 LM75 通信。

void lm75_init(unsigned char chip,signed char thyst,signed char tos, unsigned char pol) ：初始化 LM75。调用这个函数前，必须调用函数 i2c_init 初始化 I^2C 总线。调用其他的 LM75 函数前，必须先调用此函数。如果有多个 LM75 接在同一个 I^2C 总线上，则每个 LM75 都要进行初始化。最多允许 8 个 LM75 接在同一个 I^2C 总线上，地址为 0～7。LM75 被设置为比较模式，就像一个恒温器。O.S.输出脚在温度高于 tos 时激活，在低于 thyst 时恢复高阻（thyst 和 tos 的单位都是℃）。pol 设置 LM75 的 O.S.输出脚在激活时的极性。如果 pol 是 0，输出激活时的极性为低；如果 pol 是 1，输出激活时的极性为高。

int lm75_temperature_10(unsigned char chip)：读地址为 chip 的 LM75 的温度，单位是℃，其值为返回值除以 10。

（2）Dallas Semiconductor DS1621 Thermometer/Thermostat Functions（DS1621 温度计函数）：这些函数放在“..\INC”目录的“ds1621.h”头文件中。I^2C 总线函数在 ds1621.h 中自动包含。包含头文件前，必须先声明哪些口线通过 I^2C 总线与 DS1621 通信。

void ds1621_init(unsigned char chip,signed char tlow,signed char thigh, unsigned char pol)：初始化 DS1621。调用这个函数前，必须调用函数 i2c_init 初始化 I^2C 总线。调用其他的 DS1621 函数前，必须先调用此函数。如果有多个 DS1621 接在同一个 I^2C 总线上，则每个 DS1621 都要进行初始化。最多允许 8 个 DS1621 接在同一个 I^2C 总线上，地址为 0～7。除了测量温度，DS1621 函数也可以像恒温器一样工作。Tout 输出脚在温度高于 thigh 时激活，在温度低于 tlow 时恢复高阻。tlow 和 thigh 的单位都是℃。pol 设置 DS1621 的 Tout 输出脚在激活时的极性。如果 pol 是 0，输出激活时的极性为低；如果 pol 是 1，输出激活时的极性为高。

unsigned char ds1621_get_status(unsigned char chip)：读出对应地址 chip 的 DS1621 配置/状态字节。

void ds1621_set_status(unsigned char chip, unsigned char data)：设置对应地址 chip 的 DS1621 配置/状态字节。

void ds1621_start(unsigned char chip)：使地址为 chip 的 ds1621 从省电模式中退出，开始温度测量和比较。

void ds1621_stop(unsigned char chip)：使地址为 chip 的 DS1621 停止开始温度测量，并进入省电模式。

int ds1621_temperature_10(unsigned char chip)：读地址为 chip 的 DS1621 温度，单位是℃，其值为返回值除以 10。

（3）Dallas Semiconductor DS1820/DS1822 Temperature Sensors Functions（DS1820/1822 温度传感器函数）：只有商业版的 CodeVisionAVR C Compiler 才有这部分功能。这些函数放在“..\INC”目录的“ds1820.h”头文件中。单线总线的函数自动在 ds1820.h 中被包含。包含头文件前，必须先声明哪些口线使用单线通信协议与器件通信。

int ds1820_temperature_10(unsigned char *addr)：返回 ROM 码存在地址 addr 处的数组中的 DS1820/DS1822 的温度。温度的值为 0.1℃。如果有错误，返回值为−9999。如果只有一个 DS1820/DS1822，就不需要 ROM 码，指针 addr 要设为 NULL（0）。如果有多个器件，要首先读 ROM 码对每个器件进行识别，然后才能在调用 ds1820_temperature_10 时对需要的器件通过 ROM 码进行地址匹配。

unsigned char ds1820_set_alarm(unsigned char *addr,signed char temp_low,signed chartemp_high)：设置 DS1820/DS1822 的低温、高温报警温度。如果设置成功则返回 1，否则返回 0。报警温度存在 DS1820/DS1822 的暂存器 SRAM 和 EEPROM 中。用于寻址器件的 ROM 码放在 addr 指向的数组中。如果只有一个 DS1820/DS1822，就不需要 ROM 码。指针 addr 要设为 NULL（0）。DS1820/DS1822 的温度报警状态可以用 w1_search 函数发送报警搜索（Alarm Search）Ech 命令检测到。

（4）Dallas Semiconductor DS18B20 Temperature Sensors Functions（DS18B20 温度传感器函数）：这些函数为了便于 C 语言的 DS18B20 单线温度传感器实现连接，放在“..\INC”

目录的“ds18b20.h”头文件中。单线总线的函数自动在 ds18b20.h 中被包含。包含头文件前，必须先声明哪些口线使用单线通信协议与器件通信。

unsigned char ds18b20_read_spd(unsigned char *addr)：读取 DS18B20 传感器中 spd 的内容。如果成功，则返回 1；若出现错误，则返回 0。spd 的内容将被保存在由 ds18b20.h 头文件定义的结构体中。

unsigned char ds18b20_init(unsigned char *addr,signed char temp_low, signed char temp_high, usinged char resolution)：设置 DS18B20 的温度上、下限报警值，并定义了温度测量位数。如果设置成功则返回 1，否则返回 0。报警温度存在 DS18B20 的暂存器 SRAM 和 E^2PROM 中。用于寻址器件的 ROM 码放在 addr 指向的数组中。如果只有一个 DS18B20，就不需要 ROM 码。指针 addr 要设为 NULL（0）。

float ds18b20_temperature(unsigned char *addr)：返回与其序列码对应的 DS18B20 传感器的温度值。如果温度的单位为℃，则其值为返回值除以 10。如果有错误，则返回值为 −9999。在首次调用 ds18b20_temperature 函数前，必须使用 ds18b20_init 函数定义温度测量位数。

16）**实时时钟函数**

（1）Philips PCF8563 Real Time Clock Functions（PCF8563 实时时钟函数）：只有商业版的 CodeVisionAVR C Compiler 才有这部分功能。这些函数放在“..\INC”目录的“pcf8563.h”头文件中。I^2C 总线函数在 pcf8563.h 中自动包含。包含头文件前，必须先声明哪些口线通过 I^2C 总线与 PCF8563 通信。

void rtc_init(unsigned char ctrl2, unsigned char clkout, unsigned char timer_ctrl)：初始化 PCF8563。调用这个函数前，必须调用函数 i2c_init 初始化 I^2C 总线。调用其他的 PCF8563 函数前，必须先调用此函数。I^2C 总线只能接一个 PCF8563。参数 ctrl2 设定 PCF8563 的 Control/Status2（控制/状态）寄存器的初值。

pcf8563.h 头文件定义了一些宏，以便设置 ctrl2 参数：RTC_TIE_ON 置 Control/Status 2 寄存器的 TIE 位为 1；RTC_AIE_ON 置 Control/Status 2 寄存器的 AIE 位为 1；RTC_TP_ON 置 Control/Status 2 寄存器的 TI/TP 位为 1。这些宏可以用“|”操作符连在一起使用，以同时设置多个位为 1。

参数 clkout 设定 PCF8563 CLKOUT Frequency（输出频率）寄存器的初值。pcf8563.h 头文件定义了以下一些宏，以便设置 clkout 参数：RTC_CLKOUT_OFF 关闭 PCF8563 的脉冲输出；RTC_CLKOUT_1 的 1Hz 脉冲输出；RTC_CLKOUT_32 的 32Hz 脉冲输出；RTC_CLKOUT_1024 的 1024Hz 脉冲输出；RTC_CLKOUT_32768 的 32768Hz 脉冲输出；参数 timer_ctrl 设定了 PCF8563 的 Timer Control（定时器控制）寄存器的初值。

pcf8563.h 头文件定义了一些宏方便设置 the timer_ctrl 参数：RTC_TIMER_OFF 关闭 PCF8563 倒计数定时器；RTC_TIMER_CLK_1_60 设置 PCF8563 倒计数定时器的时钟频率为 1/60Hz；RTC_TIMER_CLK_1 设置 PCF8563 倒计数定时器的时钟频率为 1Hz；RTC_TIMER_CLK_64 设置 PCF8563 倒计数定时器的时钟频率为 64Hz；RTC_TIMER_CLK_4096 设置 PCF8563 倒计数定时器的时钟频率为 4096Hz。

unsigned char rtc_read(unsigned char address)：从 PCF8563 的 address 地址寄存器读出 1B。

void rtc_write(unsigned char address, unsigned char data)：写 1B 到 PCF8563 的 address 地址的寄存器中。

unsigned char rtc_get_time(unsigned char *hour, unsigned char *min, unsigned char *sec)：返回 RTC（实时时钟）的时间。指针*hour、*min 和*sec 必须指向接收小时、分钟和秒的变量。如果函数返回 1，则说明读的时间正确；如果函数返回 0，则说明供电电压太低，时间不正确。

void rtc_set_time(unsigned char hour, unsigned char min, unsigned char sec)：设置 RTC 的时间。参数 hour，min 和 sec 对应时、分、秒。

void rtc_get_date(unsigned char *date, unsigned char *month, unsigned *year)：读取 RTC 的日历。指针*date，*month 和*year 指向接收日、月、年的变量。

void rtc_set_date(unsigned char date, unsigned char month, unsigned year)：设置 RTC 的日历。

void rtc_alarm_off(void)：关闭 RTC 的闹铃功能。

void rtc_alarm_on(void)：打开 RTC 的闹铃功能。

void rtc_get_alarm(unsigned char *date, unsigned char *hour, unsigned char *min)：读取 RTC 的闹铃的日期和时间。指针*date，*hour 和*min 指向接收日期、小时、分钟的变量。

void rtc_set_alarm(unsigned char date, unsigned char hour, unsigned char min)：设置 RTC 的闹铃的日期和时间。参数 date，hour 和 min 对应日期、小时、分钟。如果 date 是 0，则这个参数将被忽略。调用这个函数后，闹铃被关闭。要调用 rtc_alarm_on 函数才能打开闹铃功能。

void rtc_set_timer(unsigned char val)：设置 PCF8563 定时器的值。

（2）Philips PCF8583 Real Time Clock Functions（PCF8583 实时时钟函数）：只有商业版的 CodeVisionAVR C Compiler 才有这部分功能。这些函数放在“..\INC”目录的“pcf8583.h”头文件中。I^2C 总线函数在 pcf8583.h 中自动包含。包含头文件前，必须先声明哪些口线通过 I^2C 总线与 PCF8583 通信。

void rtc_init(unsigned char chip, unsigned char dated_alarm)：初始化 PCF8583。调用这个函数前，必须调用函数 i2c_init 初始化 I^2C 总线。调用其他的 PCF8583 函数前，必须先调用此函数。如果 I^2C 总线上接有多个 PCF8583，必须通过参数 chip 对每个 PCF8583 都进行初始化。I^2C 总线上最多允许接两个 PCF8583，它们的地址是 0 或 1。参数 dated_alarm 设定 RTC 闹铃是日期和时间同时起作用（dated_alarm=1），还是只有时间起作用（dated_alarm=0）。调用这个函数后 RTC 闹铃是关闭的。

unsigned char rtc_read(unsigned char chip, unsigned char address)：读 PCF8583 的 SRAM 中的 1B。

void rtc_write(unsigned char chip, unsigned char address, unsigned char data)：向 PCF8583 的 SRAM 中写 1B。在向 SRAM 中写数据时，地址 10h 和 11h 存放的是年份的值。

unsigned char rtc_get_status(unsigned char chip)：返回 PCF8583 的控制/状态寄存器的值。调用这个函数时，全局变量__rtc_status 和__rtc_alarm 自动更新。变量__rtc_status 存放着控制/状态寄存器的值。变量__rtc_alarm 如果为 1，则表示有闹铃。

void rtc_get_time(unsigned char chip, unsigned char *hour, unsigned char *min, unsigned char *sec, unsigned char *hsec)：读出 RTC 的当前时间。指针*hour，*min，*sec 和*hsec 指向接收小时、分钟、秒和百分之一秒的变量。

void rtc_set_time(unsigned char chip, unsigned char hour, unsigned char min, unsigned char sec, unsigned char hsec)：设置 RTC 的时间。参数 hour，min，sec 和 hsec 对应小时、分

钟、秒和百分之一秒的值。

void rtc_get_date(unsigned char chip, unsigned char *date, unsigned char *month, unsigned *year)：读出 RTC 的当前日期。指针*date，*month，*year 指向接收日、月、年的变量。

void rtc_set_date(unsigned char chip, unsigned char date, unsigned char month, unsigned year)：设置 RTC 的日期。

void rtc_alarm_off(unsigned char chip)：关闭 RTC 的闹铃。

void rtc_alarm_on(unsigned char chip)：打开 RTC 的闹铃。

void rtc_get_alarm_time(unsigned char chip, unsigned char *hour, unsigned char *min, unsigned char *sec, unsigned char *hsec)：读出 RTC 闹铃的时间设置值。指针*hour，*min，*sec 和*hsec 指向接收小时、分钟、秒和百分之一秒的变量。

void rtc_set_alarm_time(unsigned char chip, unsigned char hour, unsigned char min, unsigned char sec, unsigned char hsec)：设置 RTC 的闹铃的时间值。参数 hour，min，sec 和 hsec 对应小时、分钟、秒和百分之一秒的值。

void rtc_get_alarm_date(unsigned char chip, unsigned char *date, unsigned char *month)：读出 RTC 的闹铃的日期设置值。指针*date，*month，*year 指向接收日、月、年的变量。

void rtc_set_alarm_date(unsigned char chip, unsigned char date, unsigned char month)：设置 RTC 的闹铃的日期值。

（3）Dallas Semiconductor DS1302 Real Time Clock Functions（DS1302 实时时钟函数）：只有商业版的 CodeVisionAVR C Compiler 才有这部分功能。这些函数放在“..\INC”目录的“ds1302.h”头文件中。包含头文件前，必须先声明哪些口线与 DS1302 通信。

void rtc_init(unsigned char tc_on, unsigned char diodes, unsigned char res)：初始化 DS1302。调用其他的 DS1302 函数前必须先调用此函数。如果参数 tc_on 为 1，则打开涓流充电。参数 diodes 设定了涓流充电时使用二极管的个数，可以是 1 或 2。参数 res 指明了涓流充电时的电阻值，res=0，表示没有电阻；res=1，表示电阻值为 2kΩ；res=2，表示电阻值为 4kΩ；res=3，表示电阻值为 8kΩ。

unsigned char ds1302_read(unsigned char addr)：在 DS1302 的 SRAM 中地址 addr 处读 1B。

void ds1302_write(unsigned char addr, unsigned char data)：在 DS1302 的 SRAM 中地址 addr 处写 1B data。

void rtc_get_time(unsigned char *hour, unsigned char *min, unsigned char *sec)：读出 RTC 的时间。指针*hour，*min，*sec 指向接收小时、分钟、秒的变量。

void rtc_set_time(unsigned char hour, unsigned char min, unsigned char sec)：设置 RTC 的时间。参数 hour，min，sec 对应小时、分钟、秒的值。

void rtc_get_date(unsigned char *date, unsigned char *month, unsigned char *year)：读出 RTC 的当前日期。指针*date，*month，*year 指向接收日、月、年的变量。

void rtc_set_date(unsigned char date, unsigned char month, unsigned char year)：设置 RTC 的日期。

（4）Dallas Semiconductor DS1307 Real Time Clock Functions（DS1307 实时时钟函数）：只有商业版的 CodeVisionAVR C Compiler 才有这部分功能。这些函数放在“..\INC”目录的“ds1307.h”头文件中。I^2C 总线函数在 ds1307.h 中自动包含。包含头文件前，必须先声明哪

些口线通过 I^2C 总线与 DS1307 通信。

void rtc_init(unsigned char rs, unsigned char sqwe, unsigned char out)：初始化 DS1307。调用这个函数前，必须调用函数 i2c_init 初始化 I^2C 总线。调用其他的 DS1307 函数前，必须先调用此函数。参数 rs 设定了 SQW/OUT 引脚上输出方波的频率，rs=0，表示输出频率为 1Hz；rs=1，表示输出频率为 4096Hz；rs=2，表示输出频率为 8192Hz；rs=3，表示输出频率为 32768Hz。如果参数 sqwe 设为 1，则允许 SQW/OUT 引脚上的方波输出。参数 out 设定了禁止 SQW/OUT 引脚上的方波输出时（sqwe=0）引脚上的逻辑电平。

void rtc_get_time(unsigned char *hour, unsigned char *min, unsigned char *sec)：读出 RTC 的当前时间。指针*hour，*min，*sec 指向接收小时、分钟、秒的变量。

void rtc_set_time(unsigned char hour, unsigned char min, unsigned char sec)：设置 RTC 的时间。参数 hour，min，sec 对应小时、分钟、秒的值。

void rtc_get_date(unsigned char *date, unsigned char *month, unsigned char *year)：读出 RTC 的当前日期。指针*date，*month，*year 指向接收日、月、年的变量。

void rtc_set_date(unsigned char date, unsigned char month, unsigned char year)：设置 RTC 的日期。

17）**E^2PROM 函数**

（1）DS2430 E^2PROM 函数：只有商业版的 CodeVisionAVR C Compiler 才有这部分功能。这些函数放在“..\INC”目录的“ds2430.h”头文件中。包含头文件前，必须先声明哪些口线与 DS2430 通信。

unsigned char ds2430_read_block(unsigned char * romcode, unsigned char *dest, unsigned char addr, unsigned char size)：从 DS2430 的 E^2PROM 中由 addr 指定地址单元读取 size 个字节块，并将它存储在目标地址为 dest 的 SRAM 单元中。若读取成功，则返回 1；否则返回 0。根据存储地址 romcode 单元中的 8B 序列号来选取 DS2430。

unsigned char ds2430_read (unsigned char * romcode, unsigned char addr, unsigned char *data)：从 DS2430 的 E^2PROM 中由 addr 指定地址单元读取 1B，并将它存储在目标地址为 data 的 SRAM 单元中。若读取成功，则返回 1，否则返回 0。根据存储地址 romcode 单元中的 8B 序列号来选取 DS2430。

unsigned char ds2430_write_block(unsigned char * romcode, unsigned char *source, unsigned char addr, unsigned char size)：将 SRAM 中起始地址为 source 的字符串中的 size 个字节写入 DS2430 的 E^2PROM 中起始地址为 addr 的目标单元中。若写入成功，则返回 1，否则返回 0。根据存储地址 romcode 单元中的 8B 序列号来选取 DS2430。

unsigned char ds2430_write (unsigned char * romcode, unsigned char addr, unsigned char data)：将单字节数据写入 DS2430 中 E^2PROM 的 addr 目标单元中。若写入成功，则返回 1，否则返回 0。根据存储地址 romcode 单元中的 8B 序列号来选取 DS2430。

unsigned char ds2430_read_appreg_block (unsigned char * romcode, unsigned char *dest, unsigned char addr, unsigned char size)：从 DS2430 用户指定的 addr 地址单元中读取 1B 块，并将它存储在 SRAM 的目标字符串中。若读取成功，则返回 1，否则返回 0。根据存储地址 romcode 单元中的 8B 序列号来选取 DS2430。如果只使用一个 DS2430，则不需要 ROM 码，指针 romcode 必须定义为 NULL（0）。如果使用了多个单线设备，则程序必须首先根据 ROM 码来识别设备。

（2）DS2433 E^2PROM 函数：只有商业版的 CodeVisionAVR C Compiler 才有这部分功能。这些函数放在“..\INC”目录的“ds2433.h”头文件中。包含头文件前，必须先声明哪些口线与 DS2433 通信。

unsigned char ds2433_read_block(unsigned char * romcode, unsigned char *dest, unsigned int addr, unsigned int size)：从 DS2430 的 E^2PROM 由 addr 变量指定地址单元中读取 size 个字节块，并将它存储在目标地址为 dest 的目标单元中。若读取成功，则返回 1，否则返回 0。根据存储地址 romcode 单元中的 8B 序列号来选取 DS2433。

unsigned char ds2433_read (unsigned char * romcode, unsigned int addr, unsigned char *data)：从 DS2433 的 E^2PROM 中由 addr 指定地址单元读取 1B，并将它存储在目标地址为 data 的 SRAM 单元中。若读取成功，则返回 1，否则返回 0。根据存储地址 romcode 单元中的 8B 序列号来选取 DS2433。

unsigned char ds2433_write_block(unsigned char * romcode, unsigned char *source, unsigned int addr, unsigned int size)：将 SRAM 中起始地址为 source 的字符串中的 size 个字节块，写入 DS2433 的 E^2PROM 中起始地址为 addr 的目标单元中。若写入成功，则返回 1，否则返回 0。根据存储地址 romcode 单元中的 8B 序列号来选取 DS2433。

unsigned char ds2433_write (unsigned char * romcode, unsigned char addr, unsigned char data)：将单字节数据写入 DS2433 中 E^2PROM 的 addr 目标单元中。若写入成功，则返回 1，否则返回 0。根据存储地址 romcode 单元中的 8B 序列号来选取 DS2433。如果使用了多个单线设备，则程序必须首先根据 ROM 码来识别设备。

参 考 文 献

[1] 陈忠平．基于 Proteus 的 AVR 单片机 C 语言程序设计与仿真．北京：电子工业出版社，2011．

[2] 陈忠平．基于 Proteus 的 51 系列单片机设计与仿真(第 2 版)．北京：电子工业出版社，2012．

[3] 陈忠平．51 系列单片机 C 语言程序设计经典实例．北京：电子工业出版社，2012．

[4] 陈忠平．基于 Proteus 的 PIC 系列单片机 C 语言程序设计与仿真．北京：电子工业出版社，2013．

[5] 马潮．AVR 单片机嵌入式系统原理与实用实践．北京：北京航空航天大学出版社，2007．

[6] 张华宇, 谢凤芹, 李跃辉．AVR 单片机基础与实例进阶．北京：清华大学出版社，2012．

[7] 张军，宋涛．AVR 单片机 C 语言程序设计实例精粹．北京：电子工业出版社，2009．

[8] 周兴华．AVR 单片机 C 语言高级程序设计．北京：中国电力出版社，2008．

[9] 杨正忠，耿德根．AVR 单片机应用开发指南及实例精解．北京：中国电力出版社，2008．

[10] 沈建良，赵文宏，贾玉坤等．ATmega128 单片机入门与提高．北京：北京航空航天大学出版社，2009．

[11] 金钟夫，杜刚，王群，徐宗完．AVR ATmega128 单片机 C 程序设计与实践．北京：北京航空航天大学出版社，2006．

[12] ATMEL．ATmega16 Data Book．http://www.atmel.com．

[13] HP InfoTech S.R.L．CVAVR USER MANUAL．http://www.hpinfotech.com．

[14] http://www.operedv.com．

[15] http://www.cepark.com．

[16] http://www.iccavr.com．

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：（010）88254396；（010）88258888
传　　真：（010）88254397
E-mail：dbqq@phei.com.cn
通信地址：北京市海淀区万寿路 173 信箱
　　电子工业出版社总编办公室
邮　　编：100036

《ATmega16 单片机 C 语言程序设计经典实例》

读者调查表

尊敬的读者：

欢迎您参加读者调查活动，对我们的图书提出真诚的意见，您的建议将是我们创造精品的动力源泉。为方便大家，我们提供了两种填写调查表的方式：

1．您可以登录 http://yydz.phei.com.cn，进入“客户留言”栏目，将您对本书的意见和建议反馈给我们。

2．您可以填写下表后寄给我们（北京市海淀区万寿路 173 信箱电子技术出版分社　邮编：100036）。

姓名：_____　性别：□　男　□　女　年龄：_____　职业：_____

电话（寻呼）：_____　E-mail：_____

传真：_____　通信地址：_____

邮编：_____

1．影响您购买本书的因素（可多选）：

□封面封底　□价格　□内容简介、前言和目录　□书评广告　□出版物名声

□作者名声　□正文内容　□其他 _____

2．您对本书的满意度：

从技术角度　□很满意　□比较满意　□一般　□较不满意　□不满意

从文字角度　□很满意　□比较满意　□一般　□较不满意　□不满意

从排版、封面设计角度　□很满意　□比较满意　□一般　□较不满意　□不满意

3．您最喜欢书中的哪篇（或章、节）？请说明理由。

4．您最不喜欢书中的哪篇（或章、节）？请说明理由。

5．您希望本书在哪些方面进行改进？

6．您感兴趣或希望增加的图书选题有：

邮寄地址：北京市海淀区万寿路 173 信箱电子信息出版分社　张剑　收　邮编：100036

编辑电话：（010）88254450　　E-mail：zhang@phei.com.cn